MicroMonitor

Embedded System Boot Platform

User Manual

Issue Date: August 6, 2007

Author: Ed Sutter

What is MicroMonitor?

MicroMonitor, or uMon, is an embedded system boot platform. It is a target-resident environment that provides the developer with a set of capabilities that enhance the development process and add to the flexibility of the application being developed. In its simplest form, it is used to speed up the early stages of embedded system development by providing a target independent mechanism for program/data storage as well as industry standard interfaces (TFTP & XMODEM) for transfer of data to/from the target. A fully enabled MicroMonitor platform includes:

- Extensible built-in flash file system (TFS)
- Support for JFFS2 and FAT
- TFTP client/server for network file transfer
- Xmodem for serial file transfer
- On-board ASCII file creation (i.e. target resident file editor)
- File-based scripts with conditional branching
- ASCII-script-driven startup options
- Command line history and editing
- UDP and RS232 based command entry
- Versatile configuration management using files
- Symbols and shell variables
- Symbolic display of variables, stack trace, runtime profiling and memory-based runtime trace
- Gdb server for application loads and post-mortem analysis
- Network host supporting ICMP and DHCP/BOOTP as a startup option
- Syslog client
- Zlib-based decompression
- Password-protected user levels
- Large API to hook the application to facilities provided by monitor

As a part of a development strategy for an organization, it provides a common base platform, an integral part of the application itself, providing the system with a core set of features that are generic in nature, and should be useable by application code regardless of the operating system chosen.

Disclaimer:

This documentation and the MicroMonitor platform discussed are provided “AS-IS”. There are absolutely no guarantees of any kind regarding this documentation or the MicroMonitor platform. Lucent Technologies has allowed me (the primary author, Ed Sutter) to make this publicly available originally from either Lucent’s Software Distribution Site (http://www.bell-labs.com/topic/swdist/) or through the book “Embedded Systems Firmware Demystified” (available from CMP publishers). Since that time, the folks at Microcross have been kind enough to maintain a web site with the most up-to-date version of the MicroMonitor package. As a result, MicroMonitor is being used on a variety of embedded systems around the world. Please report comments, questions or deficiencies in this documentation (or the MicroMonitor platform) directly to me at esutter@lucent.com.

Note that this is a “living” document. The cover page includes the issue date so check back frequently to make sure you have an up-to-date copy. The reference to uMon 1.0 simply means uMon 1.0 and beyond.

Acknowledgements:

This monitor package was written by me, Ed Sutter, and in many places both in this text and in the MicroMonitor source code itself there are references to the fact that I was the originator. While it is still reasonable for me claim to have written the bulk of the code, there have been several contributors to the package over the years. A contributor is not necessarily someone who has written code (although this is certainly appreciated!), it can be a user that reports a bug or makes a suggestion on how to improve or add to the package's capabilities and usefulness. So, as you read through this text/code, be aware that as the number of contributors increases, putting one name next to the term "author" is a miss-representation of credit. Many thanks to those who have helped out!

Finally, I need to thank a few folks for their efforts while I put this text together. My wife Lucy and son Tommy put up with me spending many weekends and evenings working through this. To them I credit my enjoyment of life. Most importantly I thank God, who has been my strength and breath throughout life, for giving me His Son Jesus Christ. Through His death and resurrection on the cross for me (and you!) 2000 years ago, I have absolute faith that I will one day walk with Him in heaven.

Now therefore fear the Lord, and serve him in sincerity and in truth… And if it seem evil unto you to serve the Lord, choose you this day whom ye will serve… but as for me and my house, we will serve the Lord.
(Judges 24:14-15)

Lift up your eyes on high and see who has created the stars,
The One who leads forth their host by number, He calls them all by name;
Because of the greatness of His might and the strength of His power,
Not one of them is missing.
(Isaiah 40:26)

He that dwelleth in the secret place of the most High shall abide under the shadow of the Almighty. I will say of the Lord, He is my refuge and my fortress: my God; in Him will I trust. He shall cover thee with his feathers, and under his wings shalt thou trust: his truth shall be thy shield.

(Psalm 91: 1-2,4)

Table of Contents

CHAPTER 1 WHAT'S NEW? 11

1.1 New to uMon 1.14 11
1.2 New to uMon 1.12 11
1.3 New to uMon 1.11 11
1.4 New to uMon 1.10 12
1.5 New to uMon1.9 12
1.6 New to uMon1.8 12
1.7 New to uMon1.7 13
1.8 New to uMon1.6 13
1.9 New to uMon1.5 14
1.10 New to uMon1.4 14
1.11 New to uMon1.3 14
1.12 New to uMon1.2: 14
1.13 New to uMon1.1: 14
1.14 New to uMon1.0: 15

CHAPTER 2 GETTING MICROMONITOR CONNECTED AND CONFIGURED 19

2.1 Applying Power to the Target System 19
2.2 Connecting to the Serial Port 19
2.3 The MicroMonitor Startup Header 20
2.4 Complete Serial Port Access 21
2.5 Configuring Network Access 21
2.6 The UDP-Based Command Interface 24
2.7 Wrap-Up 24

CHAPTER 3 BECOMING FAMILIAR WITH THE TARGET 26

3.1 Getting Comfortable with the Command Line Interface (CLI) 26
3.2 The MicroMonitor Startup Environment 30
3.3 The Startup File: monrc 31
3.4 Using Shell Variables and Symbols 32
3.5 Command Line Redirection 34
3.6 Wrap-Up 35

CHAPTER 4 FILE/DATA TRANSFER TO AND FROM THE TARGET 36

4.1 MicroMonitor's Xmodem Facility 36
4.2 MicroMonitor's TFTP Facility 39
4.3 Wrap-Up 44

CHAPTER 5 APPLICATION STARTUP 45

5.1 TFS File Attributes and Info 45
5.2 Executable, Autobootable & monrc 46
5.3 The Application Script 47
5.4 MicroMonitor's Startup Using Autobootable Files 48
5.5 Autoboot Warning 49
5.6 Compressed Executables in TFS 50
5.7 User Levels for Commands and Files 51
5.8 Wrap-Up 54

CHAPTER 6 BOOTING OFF THE NETWORK....56

6.1 DHCP/BOOTP Details: 56
6.2 DHCP Specifically... 58
6.3 Preparing the Server 58
6.4 A BOOTP Example 59
6.5 A DHCP Example 61
6.6 DHCP Coordinated with a Startup Script 62
6.7 Network Boot without DHCP or BOOTP 63
6.8 Wrap Up 64

CHAPTER 7 WRITING MICROMONITOR SCRIPTS....65

7.1 Script Invocation 65
7.2 Script-Specific Commands 66
7.3 Script Nesting 66
7.4 Scripts Calling Binary Applications 66
7.5 Example #1: cleanup 67
7.6 Example #2: ping 67
7.7 Example #3: namelist 68
7.8 Example # 4: namelist using "item" command 68
7.9 Example # 5: processing a variable number of command line arguments 69
7.10 Example # 6: why would you ever want to do this??... 69
7.11 Example # 7: startup script using subroutines, if/else and file decompression 69
7.12 Example #8: Retrieving and Displaying a Bitfield Within a Memory Location 71
7.13 Overriding the Default Command Interpreter 72
7.14 Replacing a Built-In Command with a Script 73
7.15 Wrap Up 75

CHAPTER 8 MICROMONITOR'S CONNECTION TO THE APPLICATION 76

8.1 The Monitor-to-Application Connection 76
8.2 Application-Provided Functionality 77
8.3 Application-Provided Mutual Exclusion 77
8.4 Application-Monitor Hookup: a small, single-threaded example 77
8.5 Application-Monitor Hookup: a VxWorks example 78
8.6 Application Installation on a Monitor Based System 81
8.7 Extending the Monitor's Heap 84
8.8 Use of Application-Provided Lockout for the Monitor's API 85
8.9 Wrap Up 88

CHAPTER 9 BINARY APPLICATION EXAMPLES 89

9.1 Architecture Independent Configuration 89
9.2 App #1: Embedded "hello world" 90
9.3 App #2: Applications Built Using Portions of MicroMonitor Common 92
9.4 Establishing a Stack Frame for Various CPU Architectures 95
9.5 App #3: Using MicroMonitor's CLI in Application Space 96
9.6 App #4: Hooking Up to TFS in Application Space 98
9.7 The "umon_apps/demo" Application 100
9.8 Wrap Up 100

CHAPTER 10 BUILT-IN DIAGNOSTICS AND DEBUG....101

10.1 The Application 101
10.2 Target-Resident Files Used for Symbolic Access 102
10.3 Symbolic Debugging 104

10.4 Run-time Trace 106
10.5 Default Exception Handling 107
10.6 Context-Sensitive Stack Trace 110
10.7 Post Mortem Analysis 111
10.8 Profiling Your Application 112
10.9 Heap Corruption & Memory Leak Detection 115
10.10 Breakpoints and Single Stepping… NOT 118
10.11 GDB Interface 119
10.12 Wrap-Up 119

CHAPTER 11 PORTING TO A NEW TARGET 120

11.1 First Things First 120
11.2 Getting Started 123
11.3 Directory Structure 124
11.4 The makefile 125
11.5 The config.h file: 130
11.6 Runtime Execution, FLASH or RAM? 132
11.7 The Memory Map 137
11.8 The Console (Serial) Driver and cpuio.c: 140
11.9 The Flash Driver: 141
11.10 Configuring TFS 147
11.11 The Watchdog Macro 154
11.12 Miscellaneous entries in config.h. 154
11.13 Adding a Target-Specific Command 155
11.14 Post-Port Testing 156
11.15 Wrap Up 158

CHAPTER 12 MISCELLANEOUS APPLICATION NOTES 160

12.1 Runtime Reconfiguration of TFS 160
12.2 Creating a RAM Based TFS Storage Area 163
12.3 Voluntarily Updating Your Monitor Image 164
12.4 Using an External Debugger (JTAG or similar) 165
12.5 How Do I Get the Size and/or Location of a File in TFS? 167
12.6 How Do I Intercept uMon Command Output in my Application? 168
12.7 How Do I Attach the Date to a File in TFS? 170

CHAPTER 13 TOPICS SPECIFC TO BOOTING EMBEDDED LINUX 172

13.1 Configuring Flash with uMon and Embedded Linux 172
13.2 Linux Startup Using MicroMonitor 177
13.3 Using JFFS2 and/or FATFS as Part of Your Startup Strategy 179
13.4 Using the ‘tfs’ Command at the Linux Prompt 180

CHAPTER 14 SHELL VARIABLES CREATED AND/OR USED BY MICROMONITOR 182

14.1 APPRAMBASE 182
14.2 ARGC 182
14.3 ARG'N' 182
14.4 APP_EXITONCLEANERROR 182
14.5 ARPRETRYTUNE 182
14.6 BOOTFILE 182
14.7 BOOTROMBASE 182
14.8 BOOTSRVR 183
14.9 CF_BLKSIZE 183

14.10 CMDSTAT 183
14.11 CONSOLEBAUD 183
14.12 DCLIPORT 183
14.13 DHCPDONTBOOT 183
14.14 DHCPCLASSID 183
14.15 DHCPCLIENTID 183
14.16 DHCPFLAGS 183
14.17 DHCPLEASETIME 183
14.18 DHCPOFFRFLTR 184
14.19 DHCPRETRYTUNE 184
14.20 DHCPRQSTLIST 184
14.21 DHCPSTARTUPDELAY 184
14.22 DHCPVSA 184
14.23 DSRVPORT 184
14.24 ENTRYPOINT 185
14.25 ETHERADD 185
14.26 EXCEPTION_SCRIPT 185
14.27 EXCEPTION_TYPE 185
14.28 FATFS_RD 185
14.29 FATFS_WR 185
14.30 FATFSNAME 185
14.31 FATFSSIZE 185
14.32 FATFSTOT 185
14.33 FLASH_BASE_N 185
14.34 FLASH_SCNT_N 185
14.35 FLASH_END_N 185
14.36 FLASH_DEVTOT 185
14.37 GDBPORT 186
14.38 GIPADD 186
14.39 IPADD 186
14.40 JFFS2NAME 186
14.41 JFFS2SIZE 186
14.42 JFFS2TOT 186
14.43 MALLOC 186
14.44 MCMDPORT 186
14.45 MEMSIZE 186
14.46 MONCMD_SRCIP 186
14.47 MONCMD_SRCPORT 186
14.48 MONCOMPTR 186
14.49 MONFLAGS 186
14.50 MTCRC 187
14.51 MONITORBUILT 187
14.52 NETMASK 187
14.53 NO_EXCEPTION_RESTART 187
14.54 NO_UMONBSS_WARNING 187
14.55 PCISIZE 187
14.56 PLATFORM 187
14.57 POLLTIMEOUT 187
14.58 PROMPT 187
14.59 RLYAGNT 187
14.60 ROOTPATH 187
14.61 SCR_EXITONCLEANERROR 188
14.62 SCRIPT_IGNORE_ERROR 188
14.63 SCRIPTVERBOSE 188
14.64 STRLEN 188
14.65 STRUCTBASE 188
14.66 STRUCTFILE 188

14.67 STRUCTOFFSET 188
14.68 STRUCTSIZE 188
14.69 SYMFILE 188
14.70 TFTPGET 188
14.71 TFS_PREFIX_N 188
14.72 TFS_START_N 188
14.73 TFS_END_N 188
14.74 TFS_SPARE_N 188
14.75 TFS_SPARESZ_N 189
14.76 TFS_SCNT_N 189
14.77 TFS_DEVINFO_N 189
14.78 TFS_DEVTOT 189
14.79 TFTPPORT 189
14.80 TFTPRCV 189
14.81 TFTPRETRYTUNE 189
14.82 VERSION_MAJ, VERSION_MIN, VERSION_TGT 189
14.83 XMODEMGET 189

CHAPTER 15 MICROMONITOR COMMAND SET 190

15.1 ARP 191
15.2 BRDINFO 192
15.3 CALL 193
15.4 CAST 194
15.5 CF 196
15.6 CM 197
15.7 DHCP 198
15.8 DIS 200
15.9 DM 201
15.10 ECHO 203
15.11 EDIT 204
15.12 ETHER 206
15.13 EXIT 207
15.14 FATFS 208
15.15 FLASH 210
15.16 GOSUB 213
15.17 GOTO 214
15.18 HEAP 215
15.19 HELP 217
15.20 HISTORY 218
15.21 ICMP 219
15.22 IF 220
15.23 ITEM 222
15.24 JFFS2 223
15.25 MT 225
15.26 MTRACE 227
15.27 PM 228
15.28 PROF 229
15.29 READ 230
15.30 REG 231
15.31 RESET 232
15.32 RETURN 233
15.33 SET 234
15.34 SLEEP 235
15.35 SM 236
15.36 STRUCT 237
15.37 STRACE 242

15.38 SYSLOG 243
15.39 TFS 244
15.40 TFTP 249
15.41 ULVL 252
15.42 UNZIP 253
15.43 VERSION 254
15.44 XMODEM 255

CHAPTER 16 MICROMONITOR APPLICATION PROGRAMMER'S INTERFACE 257

16.1 monConnect() 258
16.2 mon_addcommand() 259
16.3 mon_appexit() 260
16.4 mon_com() 261
16.5 mon_cprintf() 262
16.6 mon_xcrc16() 263
16.7 mon_crc32() 264
16.8 mon_decompress() 265
16.9 mon_delay() 266
16.10 mon_docommand() 267
16.11 mon_flasherase() 268
16.12 mon_flashinfo() 269
16.13 mon_flashwrite() 270
16.14 mon_free() 271
16.15 mon_getargv() 272
16.16 mon_getbytes() 273
16.17 mon_getchar() 274
16.18 mon_getenv() 275
16.19 mon_getenvp() 276
16.20 mon_getline() 277
16.21 mon_getsym() 278
16.22 mon_gotachar() 279
16.23 mon_heapextend() 280
16.24 mon_i2cctrl() 281
16.25 mon_i2cread() 282
16.26 mon_i2cwrite() 283
16.27 mon_intsoff() 284
16.28 mon_intsrestore() 285
16.29 mon_malloc() 286
16.30 mon_memtrace() 287
16.31 mon_pcicfgread() 288
16.32 mon_pcicfgwrite() 289
16.33 mon_pcictrl() 290
16.34 mon_portcmd() 291
16.35 mon_printf() 292
16.36 mon_printmem() 293
16.37 mon_printpkt() 294
mon_profiler() 295
16.38 mon_putchar() 296
16.39 mon_realloc() 297
16.40 mon_recvenetpkt() 298
16.41 mon_restart() 299
16.42 mon_sendenetpkt() 300
16.43 mon_setenv() 301
16.44 mon_setUserLevel() 302
16.45 mon_sprintf() 303
16.46 mon_timeofday() 304

16.47 mon_tfsadd() 305
16.48 mon_tfsclose() 307
16.49 mon_tfsctrl() 308
16.50 mon_tfseof 311
16.51 mon_tfsfstat() 312
16.52 mon_tfsgetline() 313
16.53 mon_tfsinit() 314
16.54 mon_tfsipmod() 315
16.55 mon_tfslink() 316
16.56 mon_tfsnext() 317
16.57 mon_tfsopen() 318
16.58 mon_tfsread() 320
16.59 mon_tfsrun() 321
16.60 mon_tfsseek() 322
16.61 mon_tfsstat() 323
16.62 mon_tfstell() 324
16.63 mon_tfstruncate() 325
16.64 mon_tfsunlink() 326
16.65 mon_tfswrite() 327
16.66 mon_version() 328
16.67 mon_watchdog() 329
16.68 mon_warmstart() 330

CHAPTER 17 HOST-BASED TOOLS 331

17.1 Building the Tools 331
17.2 Building Tools with Visual C++ 331
17.3 AOUT 332
17.4 BIN2ARRAY 333
17.5 BIN2SREC 334
17.6 COFF 335
17.7 DEFDATE 337
17.8 DHCPSRVR 338
17.9 ELF 341
17.10 F2MEM 342
17.11 FCRC 344
17.12 MACCRYPT 345
17.13 MAKE2FLIST 346
17.14 MKUPDATE 347
17.15 MONCMD 349
17.16 MONSYM 350
17.17 NEWMON 352
17.18 TNT 353
17.19 TTFTP 354
17.20 uCon 355
17.21 VSUB 356
17.22 Some Handy Host-Based Scripts… 357

Chapter 1 What's new?

MicroMonitor is a "living" project. Features are added and bugs are fixed; hence there is a need for some mechanism for keeping track of the progress. The original intent of this chapter was to introduce the changes that were a result of the initial uMon1.0 release. Since release of uMon1.0, this chapter has turned into a general "What's New?" text. The minor version number of uMon increments each time a user-visible change is made, and this section will document those changes. So, this is a "living" document; hence, check the site http://www.microcross.com/html/micromonitor.html occasionally for updates. This chapter discusses what has changed, what has been eliminated and what has been added to MicroMonitor for the new 1.0 release, and each new release following. It covers features added to uMon as well as updates made to the documentation.

1.1 New to uMon 1.14

Release 1.14 available Aug 6, 2007

- New –p option to the ttftp host command to allow the user to override the default TFTP port of 69. This applies to both the client and server.
- New host tool: mkupdate. Provides a quick and easy way to build a script that can be downloaded to a target and used to check if any files need to be updated from a TFTP server. Refer to manpage for more info.
- New DHCP shell variable: ROOTPATH. This variable is populated by the content of DHCP option #17 if the incoming DHCP message contains option 17 (root path).
- New DHCP/BOOTP shell variable: DHCPDONTBOOT. If this variable is present, then if DHCP or BOOTP transfers a file to the target, the data will not be transferred to a file in TFS and the data will not be executed. It will then be left up to the user to deal with the size of the data (stored in $TFTPGET) and the starting point of the data (stored in $APPRAMBASE).
- New –x option to the ttftp host command that tells the server to exit automatically after receiving one file.
- The ttftp client now swallows incoming TFTP_OACK opcodes from servers (response to the "TSIZE" option used by ttftp by default).
- The "heap –X" uMon command used to establish an extension to uMon's heap space, now does a check to make sure that the requested area in memory does not conflict with uMon's BSS space.
- Bug fix: In the command line interpreter when multiple commands are separated by semicolons, if a shell variable is expanded, the command line was corrupted.
- Change: In the tftp client/server, if a out-of-sequence block number is received, it is just ignored. Prior to this, if an out-of-sequence block number was received, the tftp code would respond with a TFTP_ERR opcode.

1.2 New to uMon 1.12

Release 1.12 available Mar 28, 2007

- Bug fix only seen with little-endian CPUs: an incoming TCP packet was not being processed properly. This has been fixed.
- New ability to load an executable binary image (elf,coff,aout) from outside of TFS flash space. The image can reside anywhere in memory and the command "tfs ld 0x12345678,E" (where 0x12345678 is the address at which the binary image is stored) and the image will be properly loaded. More info on this in the "tfs command" section.

1.3 New to uMon 1.11

Release 1.11 available Feb 26, 2007

- New API function: mon_timeofday().
- The "quick" clean defrag algorithm used by "tfs qclean" and also by tfsclean2.c is a little smarter. The original version of this algorithm simply copied all the active files to ram, then erased all the flash then copied all the active files back to flash; hence, removing all the dead space from the flash. To be a bit more efficient, now it will only erase the sectors of flash that are changed as a result of the update. Depending on what is in the flash and how the files are organized, this can greatly decrease the time taken to do a "qclean" and also the wear and tear on the flash device.

- Found (and fixed) a bug in uMon's application-accessible packet transfer interface… The two functions (mon_sendenetpkt() & mon_recvenetpkt()) allow an application to use uMon's ethernet packet interface without knowing anything about the actual ethernet device driver. The problem was caused by the fact that the application can use various uMon API calls that may have underlying code that polls the ethernet interface. When running just uMon, this is appropriate; however, when running an application on top of uMon that uses the packet interface, this causes confusion. As a result, there are two "special case" uses of mon_sendenetpkt(char * pkt, int len)...
  mon_sendenetpkt(0,0) tells the underlying uMon-based ethernet driver NOT to poll the ethernet interface internally.
  mon_sendenetpkt(0,-1) tells the underlying uMon-based ethernet driver to go back to polling the ethernet interface as normal.
- The above issue was discovered as a result of a rewrite of a simple UDP-based application that uses these two API calls. The umon_apps/udp directory is now much more robust and includes ARP and ICMP thus making it a more complete network host. Obviously this is just a simple polled interface for UDP-based data transfer; however, if that's all you need, then this should work on any target that runs uMon out-of-the-box.
- More work done on the Blackfin BF537 port, plus a new **AS3DEV** port (Xilinx's Microblaze on Avnet Spartan3 Development Kit board). Refer to the appropriate directories within the distribution for more information.

1.4 New to uMon 1.10

Release 1.10 available Nov 1, 2006

- Documentation: Lot of updates to this document, to include a chapter dedicated to booting embedded linux.
- New port to Analog Devices Blackfin processor, particularly the BF537 on the ADDS-BF537-STAMP board.
- New command: 'struct' used to overlay a structure onto memory and modify that memory based on the members of the structure. This command eliminates the need to rebuild the monitor if the kernel/bootloader interface changes. Refer to the 'struct' man page (section 15.36) for more information.
- Cleanup: Some of the newer tools would not build under umon_main/host when building with VCC=TRUE. This issue has been resolved and tested with VCC6.0.
- Bugfix: The code used to match on a GOTO/GOSUB tag would break in cases where tags in the file contained other tags as prefixes. For example, "goto ABC" would match on "# ABC" or "# ABC_DEF" or "# ABCDE" etc.. As a result, the "goto ABC" command would jump to the matching tag closest to the top of the file. This is changed so that the match is further qualified by whitespace or a colon after the tag in the file. This still allows additional text to be on the tag line, but requires a known delimiter on the endpoint of the tag.

1.5 New to uMon1.9

Release 1.9 available October 4, 2006

- Upon completion of each new file received by uMon's TFTP server, the shell variable TFTPRCV is populated with the size of the transfer in bytes.
- The file editor (edit command) will automatically append a linefeed to the end of an ASCII file that doesn't have one.
- New port to the Altera Nios processor, contributed by Graham Henderson.
- Thanks to a lot of good discussion on the MicroMonitor mail list, and a lot of work from Graham Henderson, uMon now has FAT filesystem code based on the free DOSFS library by Lewin Edwards (available from http://www.zws.com/products/dosfs/) along with a few new commands: "fatfs" , "cf" and "sd". This supports a new model for uMon to deal with externally accessible file systems. A detailed discussion of this is in the file umon_main/target/common/fatfs.c plus additional text can be found in the fatfs and cf command manpages.

1.6 New to uMon1.8

Release 1.8 available August 1, 2006.

- New 'sec' comparison for use with 'if'. This is a case-insensitive version of 'seq' (string equals).

- New section (7.14) discussing how a built-in command can be replaced by a TFS executable (script or binary).
- Fixed a bug in JFFS2 command as it was not properly dealing with truncated files.
- Improved "tfs cfg" command so that it is no longer a "one-shot" deal. The configuration can be reprogrammed as needed. Plus, it now works with versions of uMon that are relocated to and execute out of RAM space. Use of this new feature is covered in section 12.1. This does require a change in the port-specific code, refer to section 11.10.4 of the porting chapter for details on this.

1.7 New to uMon1.7

Release 1.7 available July 17, 2006

- The –e option of the set command had a syntax error, requiring an extra (ignored) argument for the command to be properly processed. This is fixed.
- The command interpreter will now totally ignore a command whose user level is higher than the currently active level. Previously, a warning was displayed indicating that the command was at a higher user level.
- To allow a TFS-based executable to seemlessly replace a built-in command, a built-in can be turned off using the "ulvl" command (refer to 'ulvl' command for more information). Plus, the "help" command will now check for the presence of the command within the built-in list, and if not found, it will look for the command in TFS, and run the command with the first (and only) argument being "help" (refer to help command for more information).
- New JFFS2 command supports listing and reading files out of JFFS2 formatted flash space. This command can be integrated into uMon as a built-in or built as a standalone TFS-based executable under umon_apps/jffs2. Refer to section 15.24 for more information.
- Eliminated the –x option in the TFS command because it is redundant with the "exit on error" script runner now.
- Bug fix: tfsadd() returned a misleading error message if the reason for the failure was due to running out of DSI space.
- New –e option to exit. This allows a script to launch another executable, but *after* the script terminates. This allows that script to be removed by the executable that it launched.
- New cache functionality for ports: disableIcache() & disableDcache() both set up as wrappers to the underlying port-specific code. Also, the flushDcache(addr,size) and invalidateIcache(addr,size) should now process addr=0, size=0 to mean flush (or invalidate, if icache) the entire cache.

1.8 New to uMon1.6

Release 1.6 available Feb 23, 2006

- The windows-only tool, uCon, is now available for download at the Microcross site. This tool is part of the MicroMonitor package but not distributed with the uMon source tarball because it is a VCC-based application. This is an "embedded developers" replacement for Hyperterminal, has a LOT of features specific (but not limited to) embedded systems development, all of which are documented in the help text that comes with the tool. There's a brief manpage included in this document, but you're better served by the built-in help text. Check it out!
- Added a few new sections to the "Miscellaneous Application Notes" (Chapter 12), and have started a set of "How Do I…" sections within that chapter that will be based on interaction I have with users.
- New host-based tools:
    - tnt (section 17.18): basic terminal connectivity with telnet server backend so that multiple users can simultaneously share a UNIX tty (serial port).
    - make2flist (section 17.13): convert the output of GNU make to a list of files useful with tools like ctags, cscope, etc..
- The READ command supports a variable prefill (-p) and no-echo (-n) option. Refer to section 15.29 (read man page).
- The PM command now supports logical operations (AND, OR & XOR). Refer to section 15.27 (pm man page).
- The command line now supports multiple commands per line with each command separated by a semicolon. Also, if the very last character on the line is a back arrow (<), then that line will be looped until a character is detected on the console. Refer to section 3.1.2 for more details.

- New API function called mon_portcmd(int cmd, void *arg). The purpose of this API is to support a port-specific call. A feature that is only applicable within a particular port. Refer to manpage for more details.
- Several of the general purpose miscellaneous functions have been pulled out of common and put into a new glib directory so that they can be rebuilt as a library. This allows the monitor to shrink a bit if various functions are not used when features are not included.
- Two additional INCLUDE_XXX macros are now required in the config.h file to provide additional configurability to the monitor build; hence a potentially smaller footprint: INCLUDE_ICMP & INCLUDE_USRLVL. See the porting chapter for more information.
- New hook added to insert port-specific code into the startup point of uMon just before the TFS autoboot function is called: PRE_TFSAUTOBOOT_HOOK(). See the porting chapter for more information.

1.9 New to uMon1.5

Release 1.5 available Nov 2005

- Prior to 1.5, if the destination filename seen by uMon's TFTP server started with a '$', then the server would attempt to replace the $STRING ("STRING" can be any shell variable name) with the content of a local shell variable. If the variable was not in the environment at the time, then the destination was just left as $STRING. This has been changed so that the server will generate a TFTP error back to the client if at the time of the transfer there is no valid shell variable.
- Bug fix: The tftp server can be disabled/enabled using "tftp off/on". If the tftp client is used when the server is disabled, it re-enabled the server, and left it enabled. This is fixed so that the client can still run without the need to enable the server; hence, if it is disabled, it will remain disabled during and after the client-invoked transfer.
- The commands "flash info" and "tfs stat" now generate a set of shell variables that contain snippets of the information displayed by the command. Refer to the command documentation for more information.
- The "flash erase" command now takes addresses or sector numbers as its argument.

1.10 New to uMon1.4

Release 1.4 available Oct 2005.

- The tfscheck() function (used by tfsctrl(TFS_CHECKDEV)), now accepts a NULL TDEV pointer to indicate that all TFS devices are to be checked.
- Fixing a fix: As a result of some changes made to the generic flash code, the address used by Xmodem –B for determining last sector to be burned had to be decremented by 1.

1.11 New to uMon1.3

Release 1.3 available as of Sept 2005.

- Bug fix: uMonInRam() function re-write.
- Bug fix: TFS ramdev device was lost after mon_appexit().
- Bug fix: TFS ramdev naming conflict could occur between device and file.

1.12 New to uMon1.2:

Release 1.2 available as of August 2005.

- The new tfs subcommand 'qclean' (i.e. quick clean) will defragment TFS-owned flash without the powersafe overhead. This is much faster; however, should only be used in cases where it is very unlikely that a power hit or reset will occur to interrupt the defrag process because if it does, files will be corrupted.
- The flash command is in the process of being re-written so that there is no need for the user to be aware of the possibility of multiple flash banks. This is only noticeable in systems that have more than one bank of flash.
- The new 'A' option to the call command builds the argument list that is used by mon_getargv().
- The new '-i' option to moncmd (host tool) that puts it in an interactive mode (similar to netcat).

1.13 New to uMon1.1:

Release 1.1 available as of August 2005..

- The 's' & 'S' options of the pm command now automatically create a STRLEN shell variable that will contain the size of the string created.

1.14 New to uMon1.0:

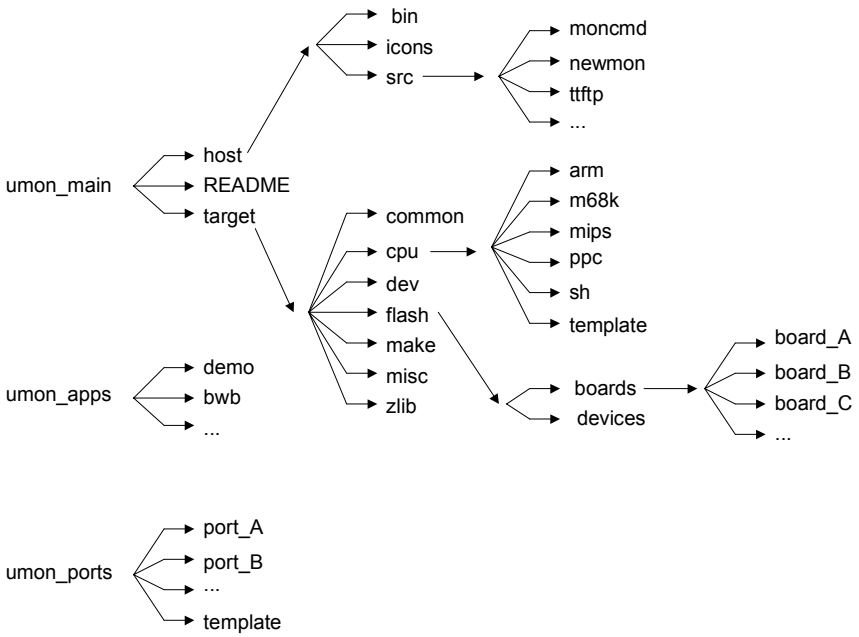
Originally this document was based on the new uMon1.0 release (around June 2005), as a result, the features added to MicroMonitor that motivated the uMon1.0 release are substantial…

1.14.1 First 'Numbered' Release

Release 1.0 of uMon (uMon 1.0) is the first "numbered" release of the MicroMonitor package. Prior to uMon1.0, MicroMonitor was distributed as a tarball, and the build-date reflected the time of creation (cross-compilation & linkage) by the individual maintaining their local copy of the source tree. Recompilation of the code, even with no change made, would cause the build-date to be updated. The build date will still follow this course; however, there is now a version number that will more accurately represent changes made to both the target-independent and target-specific code. The numbering scheme is 3 digits: A.B.C, where 'A.B' represents the port-independent code release and 'C' represents the port-specific portion of the code. This allows the port-specific code to change without affecting the release number of the port-independent code (and visa-versa).

1.14.2 New Directory Structure

A typical uMon source distribution will contain three top-level directories: *umon_main*, *umon_ports* and *umon_apps*. The target-independent portion of the source tree, *umon_main*, now contains the source for both the target-resident monitor and the host-resident tools used in conjunction with the monitor. The port-specific portion of the source tree, *umon_ports/port_name*, is a single directory per port. A new template directory provides a skeleton set of files needed to build a "start-from-scratch" port to a new target. Each of the files that need source code under the template directory have detailed comments for each function that needs code.



• **Figure 1: uMon 1.0 Source Directory Structure**

The build process for a given port now starts with the host-based tools. The top-level makefile for building the tools on Linux, Solaris or Windows (Cygwin or VCC) is found under *umon_main/host*, and a README file in that directory explains the make command line (dependent on the host OS). Once the tools are built and installed under *umon_main/host/bin*, the port-specific code can be built. The port-specific directory contains only the code needed to hook uMon's target-independent code to the system.

There is no longer a port-specific *app* directory, rather a generic *umon_apps* directory that is peer to *umon_main,* and provides a common example for each of the CPU architectures (as well as other generically useful examples). The makefile in this directory explains how to build for each of the major CPU architectures currently supported by uMon. The application includes a few examples on hooking up to the monitor's API,

establishing a simple stack area independent of the monitor and also a self-inflicted exception that is used to demonstrate the monitor's ability to dump a stack trace.

1.14.3 New Format for the Makefile

The uMon makefile has changed drastically. Dependencies are generated automatically, and it just takes better advantage of the capabilities of GNU make. There will be a lot more discussion on this in the porting chapter.

1.14.4 Less Target-Specific Code

For those who are familiar with MicroMonitor, you'll notice that there is less target-specific code under the port directory. The 'mstat' command is gone, the output of mstat is now through "help –i" and the baud rate is now changed with "set –b". There's no longer a port-specific main.c, that code was 99% common for all targets, so it has been pulled back into the *umon_main/target/common* directory.

1.14.5 Smaller Footprint If Needed

Three additional blocks of code can now be conditionally removed by INCLUDE macros in config.h: shell variables (INCLUDE_SHELLVARS), memory allocation (INCLUDE_MALLOC) and descriptive help text (INCLUDE_VERBOSEHELP). This allows the flash footprint of a minimal system (including either XMODEM or TFTP for data transfer) to fall below 64K.

1.14.6 Elimination of Some INCLUDE_XXX Macros (and their underlying code)

A few features of uMon have been removed. Each of the INCLUDE_XXX macros are checked by the inc_check.h header file to make sure that the user has removed them… INCLUDE_DEBUG, INCLUDE_UNPACK, INCLUDE_PIO

1.14.7 New and/or Deleted Commands and Command Options

Several commands have been modified, added and/or deleted…

- **ARGV**: This command was used within a script to populate shell variables ARGC and ARG'N' with the command line argument count and list. As of uMon1.0, the ARGC and ARG'N' shell variables are automatically created for each script invocation; hence, no more argv command.
- **CALL:** A new –A option allows the call command to populate the argument list used by mon_getargv(). This is something that was previously handled with the argv command. This is in uMon1.2.
- **DHCP**: There's a new '-r' option that allows the dhcp command to walk through the retry timeouts the same way it would if the protocol was invoked by the IPADD=DHCP setting in monrc.
- **ETHER**: New '-d' option to enable/disable driver debug mode (useful for Ethernet driver development).
- **FLASH**: The 'trace' subcommand now takes a numerical value instead of "on" and "off". As of uMon1.2, the subcommands within flash are becoming less dependent on the bank number. If your target only has 1 flash bank, then this has no significance.
- **LET**: This command is eliminated. The expression evaluation capability is now handled by the 'set' command. The syntax is essentially identical… set VAR=EXPR. The command line parsing within 'set' handles this by processing the expression if the single argument following 'set' contains an equal sign.
- **MSTAT**: This command has been eliminated. The two primary uses for mstat were the –b option to redefine the console's baud rate and the output of the monitor's state. The "–i" option to the "help" command dumps what used to be dumped by mstat, and the "-b" option to "set" command allows the user to redefine the baud rate.
- **MT**: The memory test command has a few new options: -S & -C. The –C option supports the ability to run a crc32 across a block of memory. The –S option allows the user to automatically detect how much memory is installed on the target.
- **PM**: The –s and –S options now create a STRLEN shell variable that will contain the size of the string created. Also, "backslash characters" (\r, \n & \t) are processed correctly.
- **PIO**: The INCLUDE_PIO macro is eliminated; hence, the default PIO command is no longer supported. It wasn't supported on most CPUs anyway. Note that if your target has a pio command, it can simply be put under the target-specific command table extension files xcmdtbl.h and xcmddcl.h.
- **SYSLOG**: This is a new command that allows uMon scripts to communicate with remote hosts over UDP. By default, the UDP port is SYSLOG (514); however, the port number is configurable; thus allowing the command to be a general purpose mechanism for UDP-based communication with a remote system. This command, along with the new MONCMD_SRCIP & MONCMD_SRCPORT shell variables, allow uMon scripts to interact programmatically with a remote host via UDP.

- **TFS**: The tfs command has a few new operations:
  "ramdev" which allows the user to dynamically create a temporary TFS device in volatile RAM space;
  "cfg" which allows the user to "one-time reconfigure" the flash space allocated to TFS.
- **TFTP**: The TFTP command's use of –v & -V (verbosity levels) has changed. The –v option tells the TFTP client to update a 'ticker' on the console each time a new data packet arrives. The –V options tells the TFTP client to dump state information per block.

1.14.8 Shell Variable Changes

- **ETHERNET_DEBUG**: If set to TRUE (or anything), then the driver will have its debug flag set. This is Ethernet driver specific.
- **MONCOMPTR**: This contains the value used to hookup the application to the monitor. Previously this was only available as output from the mstat command.
- **MONCMD_SRCIP**: Loaded with the IP address of the remote host that issued the most recent remote command via the moncmd UDP interface.
- **MONCMD_SRCPORT:** Loaded with the IP address of the host that issued the most recent remote command via the moncmd UDP interface.
- **PATH**: To allow the user to organize executables under TFS name space, the PATH shell variable provides a similar function as it does on UNIX. It is used as a prefix to executables that may be stored with a "directory-like" name.
- **STRLEN**: This shell variable is generated by the –s and –S options of the pm command to reflect the size of the created string.
- **VERSION_MAJ, VERSION_MIN, VERSION_TGT**: Contains the major, minor and target-specific version number of the running monitor. Putting each of these into a separate shell variable allows the user to easily construct a script that can be version specific.

1.14.9 No More Per-Port "app" Directory

Prior to uMon1.0, each port had its own companion "app" directory that provided a simple example of how to build an application for that particular port. This turned out to be 95% duplicate code for each port; hence, instead of duplicating this for each port, there is a new "umon_apps/demo" directory that is set up to allow the user to build a small demo application for any port. It can then be copied to local user space and turned into a real target-specific application as needed.

1.14.10 Additional Stack Sanity

Two new functions are added to MicroMonitor for uMon1.0 that support improved stack frame validation of a running uMon. The function stkinit() can be called prior to start() (in reset.S code) to initialize the stack area so that it can later be analyzed by the function stkusage(). The function stkusage() is now called as part of the "help –i" command output. It will dump information about the running stack, including the address and size of the stack, plus the percentage of the allocated stack space that has been used. This is useful for debugging uMon ports or for verification of adequate stack size allocation. Refer to the template port source code reset.S for example code that calls stkinit() prior to start().

1.14.11 Enhancements to TFS

Three Distinct Memory Types

TFS now has three different memory types: FLASH, NVRAM and RAM. Prior to uMon 1.0, the NVRAM and RAM types were considered the same. The tfsdev.h file allowed the user to specify any number of FLASH and/or RAM based TFS "devices"; however, all had to be declared at build time. There is now a differentiation made between RAM and NVRAM to make it easy for a user to temporarily create a RAM (i.e. volatile) based TFS device on the fly. The NVRAM type of memory still requires configuration via tfsdev.h; but a new TFS sub-command, "ramdev", allows the user to turn a block of volatile RAM into a new, temporary TFS device on the fly. Since this is in volatile ram, it is not protected by TFS's powersafe defragmentation algorithm; however, it does provide a convenient means of establishing a temporary TFS device.

Monrc Protection and Abortable Autoboot Option

The automatic execution of the monrc file has always been non-interruptible. This is done intentionally so that protection levels can be established at startup without any chance that the step will be skipped or omitted. The downside to this is that folks occasionally put startup scripts or even ELF executables in the monrc

file. If you don't know what you're doing, this can cause the boot process to hang; hence, requiring that the boot flash be reprogrammed by an external means (JTAG or similar). Two changes have been made in uMon 1.0 to ease this pain… During the execution of the monrc file, only a subset of uMon's command set is available, plus while in monrc execution, no other binary executable can be run (scripts can still be run from monrc). This limits the scope of the monrc file to be used, as it should be, for simple environment setup. Also, for those applications that don't need the un-interruptibility of the monrc file, if TFS_AUTOBOOT_ABORTABLE is defined in config.h, then the monrc file is interruptible.

Configurable "Make-Before-Break"

TFS's philosophy has always been "make-before-break" regarding the replacement of a file for a newer version of a file with the same name. This generally makes sense because you don't want to remove a file from flash until after you know that the replacement file is secure. The one negative side to this is that you are forced to have twice as much flash as your file actually needs for storage. If this is unacceptable, the user can now define TFS_DISABLE_MAKE_BEFORE_BREAK in config.h so that the file is removed prior to installation of the new file of the same name.

Configurable Auto-Defragmentation

When TFS's flash space fills up with a mixture of active and deleted files, an automatic defragmentation will be done. In some cases this auto-defrag is undesirable, so the user can define TFS_DISABLE_AUTODEFRAG in config.h to disable the automatic running of "tfs clean" when needed; thus forcing the user to manually run "tfs clean" to defragment the flash to recover space.

New TFS commands: cfg, ramdev

TFS can come "out-of-the-box" configured to use some pre-defined block(s) of flash and possibly RAM. In most cases, TFS simply spans across all FLASH not used by the monitor binary. Usually this is adequate; however, in some cases, a portion of the flash is used by TFS and the remaining flash is used by the application for another flash file system or just for raw flash storage space. Prior to uMon1.0, if you didn't have uMon source code, then you couldn't change the configuration on your board. The "cfg" command provides the user with the ability to do a one-time re-configuration of TFS's use of on-board flash. So, if you buy an evaluation board running uMon1.0 and you need to use some of the flash in a different way than the default configuration, you can redefine it using the "tfs cfg" command. The "tfs cfg" command modifies uMon's internal TFS configuration space in flash, so this is a one-time deal; however, if you have the image binary, you can reload the monitor and then re-do the "tfs cfg" once again. Similarly, the "ramdev" command allows the user to allocate a block of RAM space as a secondary (i.e. temporary file storage area within TFS). The allocated block of RAM is then just another TFS device; hence, the user can place files in that space and treat it like any other TFS device. The only exception to this is that it is volatile through a reset, that is, it is temporary file space.

Chapter 2 Getting MicroMonitor Connected and Configured

This chapter runs through the very basic startup of MicroMonitor's connection to the host. A typical system running MicroMonitor out of the box may need to be configured with a MAC address and HyperTerminal[1] must be configured to properly attach to the target.

2.1 Applying Power to the Target System

This procedure is dependent on your target system. Different types of boards have different types/styles of power supplies and connectors. Do not assume that a physical match between connectors implies the correct power supply. In many of today's wall-mount power supplies the same physical connector can supply different voltage types (i.e. AC/DC), voltage levels (i.e. 3.3v, 5v, 9v, 12v, etc..) and current output capabilities. Even with all else the same, the polarity on the connector may be opposite that of your target hardware. The bottom line is, always verify that you are using the properly matched power supply to your system and, throughout all of this, be aware of static discharge while handling the hardware.

The _____ system uses a _____ volt power supply.

2.2 Connecting to the Serial Port

This procedure in this section is also dependent on your target system and the host you are connecting up to. This text assumes the host is a PC. A standard PC COM port is a DTE (**D**ata **T**erminal **E**quipment) and wants to be connected to a DCE (**D**ata **C**ommunications **E**quipment). A target's serial port may be configured as DTE or DCE. To connect a DTE to a DTE, a NULL-Modem connector must be installed in the path between the two. This, among other things, makes sure that XMT on the PC is tied to RCV on the target (and visa versa).

All standard uMon console ports are configured for 8N1 (8-bit characters, no parity and 1 stop bit). Most uMon targets are configured to boot up at 19200 or 38400 baud; however this is target specific. Refer to the target documentation for this information.

The _____ system console boots up at _____ baud and uses serial port _____.

Upon establishing this physical connection, the board can be reset or powered up and the MicroMonitor header will be displayed shortly after. Note that depending on the target system, varying degrees of initialization of periphery must take place, so the output of the header may occur anywhere from immediately to a few seconds after reset.

If there is no communication between your computer and the target system after power-up, verify the following…

- Are you physically connected to the correct host COM port?
- Are you physically connected to the correct serial port on your target system (some targets have more than one connector)?
- If needed, is there a NULL modem installed?
- Verify baud rate, parity and stop bit as described above.

If all of these tests are verified and there is no connectivity between your computer and target system, then contact the manufacturer.

[1] This documentation refers to HyperTerminal because it is usually a standard part of a PC's operating environment; however, MicroMonitor has a 'companion' terminal emulator called 'uCon', which can be downloaded from the MicroMonitor website as a separate Windows-only installation.

2.3 The MicroMonitor Startup Header

Immediately after power is applied, the installed MicroMonitor firmware begins execution. It initializes some of the basic IO of the board, scans the flash that is used by the flash file system (TFS) and then dumps a MicroMonitor header to the console serial port. The following listing shows the output ..

```
MICRO MONITOR 1.0.1
Platform: Cogent CSB360
CPU: MCF5272
Built: May 20 2005 @ 10:54:20
Monitor RAM: 0x000400-0x01c128
Application RAM Base: 0x01d000
MAC: 00:60:1d:02:0b:87
IP: 192.168.1.110
uMON>
```

• **Listing 1: MicroMonitor Startup Header**

Depending on the state of your target, one of two different startup headers should have been displayed. If this is the very first time your system is out of the box, then it might need to be initially configured with a MAC address to allow it to eventually communicate over Ethernet. The target system may or may not have a pre-configured MAC address installed, so the first interaction may be:

```
MAC address must be configured.
The system's MAC address is a 6-digit, colon-delimited string
(for example: 12:34:56:78:9a:bc).  It must be unique for all
Ethernet controllers present on a given subnet.  MAC addresses
are often allocated by a product vendor to prevent duplication,
and are frequently documented on decals or other materials
provided with the product.  The following prompt allows you
to specify the MAC address for this device.
Use backspace if the printed default needs modification...

Enter MAC address (xx:xx:xx:xx:xx:xx):
00:23:31:55:
```

• **Listing 2: MAC Address Entry Prompt**

If so, this indicates that the board has a "fresh" monitor installation, meaning that the monitor has not stored away a user-defined MAC address in its flash space yet. At this point, you can backspace over the digits and re-enter a different MAC address. Depending on the target, a partial MAC address may be provided. This usually means that the target is from a vendor that has purchased a block of MAC addresses and the remaining digits represent the block owned by that vendor. Once a complete MAC address has been established, hit ENTER. The monitor will respond with the following prompt:

```
Configuring '00:60:1d:02:0b:87' as MAC address, ok? (y or n)
```

Respond with 'y' or ENTER to accept the setting; else, type 'n' to re-enter the MAC address. Once the MAC address is entered and approved, the target will respond with something like:

```
MAC address burned in at 0xff80000c
```

(address shown will vary from target to target) and then the normal startup header (as shown in Listing 1) will be displayed. With this MAC address burn-in step completed, subsequent resets and power cycles will not generate this same interaction; rather, the monitor will use the burned-in MAC address as part of its basic configuration and the header of Listing 1 will be the first output from the target[2].

[2] MicroMonitor can be configured to not display this header at startup. Refer to the description of the shell variable MONFLAGS to do this.

Assuming the above connection has been established and the MAC address has been entered, the target should have generated a MicroMonitor startup header similar to what is shown in Listing 1. The header contains some basic information about the platform, including the time and date of the monitor firmware build and the firmware version number. The two main points of interest in this header are the network address and the RAM memory map. Right now we are working only with a serial port, so aside from configuring the MAC address, we will hold off on further IP configuration for now.

2.4 Complete Serial Port Access

At this point you've set up the target and verified that the connection from target to host is working. To really be certain of complete serial port connectivity, you need to type a few characters in at the uMON> command line interface (CLI) and make sure that the target responds. Type the command "help". If you see the 'h', then you can already rest easy because this means that the target received the character and echoed it back to you; hence, two-way connectivity is working![3] The output of "help" looks something like this:

```
uMON>help

Micro-Monitor Command Set:
arp         call        cast        cm          dhcp        dis
dm          echo        edit        ether       exit        flash
fm          gosub       goto        heap        help        ?
history     icmp        if          item        mt          mtrace
pm          prof        read        reg         reset       return
set         sleep       sm          strace      ulvl        tftp
tfs         unzip       xmodem      version
uMON>
```

The exact set of commands displayed depends on how the target's MicroMonitor build was configured; however, the majority of these commands are pretty standard.

2.5 Configuring Network Access

Now we need to establish the ability to talk to the target over an IP network. Physically, there are two likely configurations:

Direct Connect: a peer-to-peer connection from the back of your PC to the RJ45 Ethernet connector on the target. This requires that the Ethernet cable be a "cross over" type and makes it impossible to have any additional devices connected to your PC.

Through a Hub: an Ethernet cable from the PC to a hub, and a second cable from the hub to the target. In this case, the cables are not crossover type, and here it is possible to have other devices on the network (through the hub).

Either configuration is acceptable, it just depends on what you have available. While it is nice to have the option to add additional devices on the hub, using the peer-to-peer configuration eliminates some of the complexities if there is a need to diagnose a problem with the connection[4] (i.e. the fewer the pieces, the easier the puzzle). Regardless, the important thing is that the physical connection is established.

Before the PC can communicate with the target over IP, the MicroMonitor firmware needs to be configured with some network address information:

IP Address: this is the address assigned to the target. MicroMonitor assumes this value will be stored in the shell variable IPADD.

[3] If you don't see any characters when you type, then the problem may be in the COM port, the cable or the target hardware.
[4] To keep things simple at the moment, if you are working with a hub and you have the option to do it, disconnect all other devices from the network so that your PC and the target are isolated in their own network space.

Network Mask: this is the IP mask that tells the target the size of the local network. This is important for the device to know when it needs to use the gateway to communicate to devices that are outside the local network. MicroMonitor assumes this value will be stored in the shell variable NETMASK.

Gateway IP Address: the address of the device on the local network that provides access to the rest of the network (devices outside the local network). MicroMonitor assumes this value will be stored in the shell variable GIPADD.

If you are working standalone with a PC, then you can configure an IP address to be on the same subnet as the PC. Use the "ipconfig" command at the DOS prompt to retrieve your PC's network configuration, and from that you can determine what values to use on the target. Use the same network mask and gateway IP address as is used by your PC. Establish a unique IP address that is on the same network (based on network mask). It is beyond the scope of this document to go much further regarding how you determine what IP/NETMASK/GIPADD values to use. This is very dependent on your network configuration. If you are working in a shared environment, it is probably best if you see your system administrator.

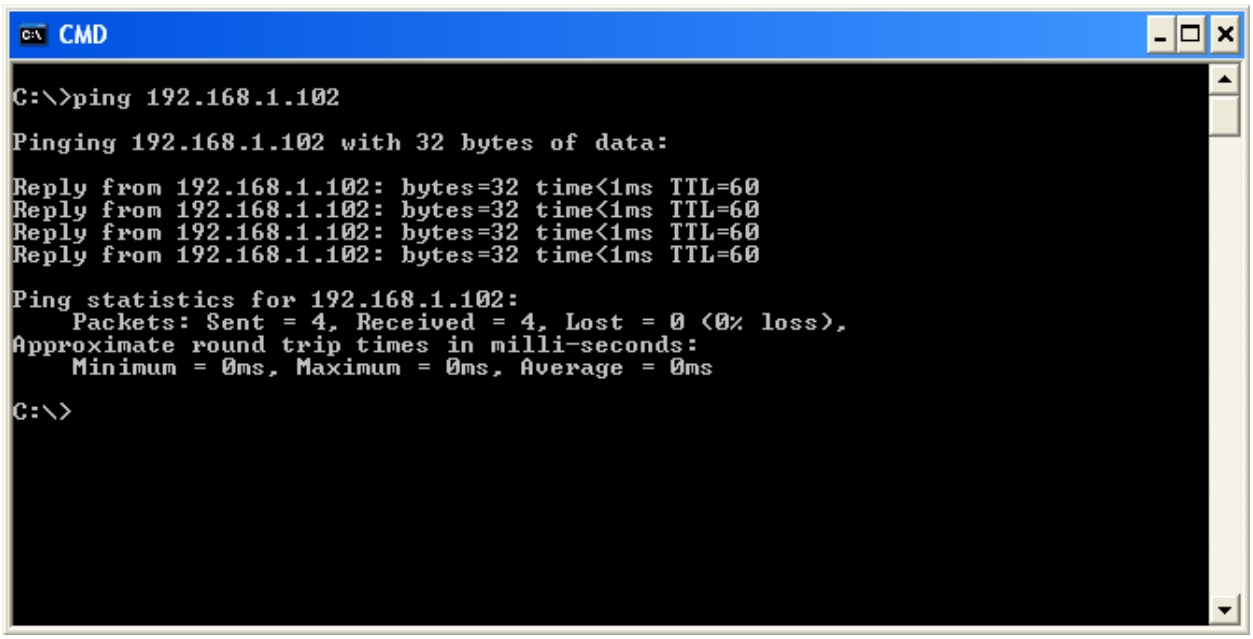
Without getting into too much detail (there will be plenty of that later) on the command syntax at this point, just issue the following commands at the uMON> prompt...

```
set -c
set IPADD 192.168.1.102
set NETMASK 255.255.255.0
set GIPADD 192.168.1.1
set -f monrc
reset
```

We'll discuss each of these commands in greater detail as we progress, for now just make sure you type in each command correctly. Note that this example uses 192.168.1.102 as the IP address and 192.168.1.1 as the gateway IP address, substitute actual network values, don't use the ones specified. This set of commands configures the environment as it is needed for network connectivity, plus it stores the environment away in a file called "monrc"[5] so that the target can be reset and the established environment will automatically be installed. The output after the "reset" command should be a startup header similar to the one in Listing 1, but now the IP address shown in the header should match the value established by the "set IPADD" command above.

Now you should be able to go to a DOS window on the PC and "ping" the target. The command "ping 192.168.1.102" (where the appropriate IP address is used) should cause the target to reply. The output at a DOS window should be similar to that of Figure 2. The "ping" command is actually an ICMP Echo request. Typically, the ping command will generate more than one request; hence, you see more than one reply.

[5] The "monrc" file is a special script that MicroMonitor looks for when it boots up. It is similar to "autoexec.bat" in DOS or ".profile" in Unix. More details are coming up.



• **Figure 2: Ping Response**

At the target, the Ethernet traffic is not seen at the serial console unless verbosity is enabled. Lets try the ping one more time, but first issue the command "ether –V" (capital V) at the uMON> prompt. Note the output (see Listing 3). The "-V" option tells the Ethernet driver in MicroMonitor to dump all traffic to the serial port. This obviously slows things down a bit, but it can be useful when debugging a new driver.

```
INCOMING PACKET (60 bytes):
  00 60 1d 02 0b 87 00 e0   18 97 62 e4 08 00 45 00   .`........b...E.
  00 3c 7a 81 00 00 80 01   3c 25 c0 a8 01 64 c0 a8   .<z.....<%...d..
  01 66 08 00 4a 5c 02 00   01 00 61 62 63 64 65 66   .f..J\....abcdef
  67 68 69 6a 6b 6c 6d 6e   6f 70 71 72               ghijklmnopqr
  Destination Host = 00:60:1d:02:0b:87
  Source Host =      00:e0:18:97:62:e4
  IP:  vhl/tos  len    id     offset  ttl/proto  csum
       x4500    x003c  x7a81  x0000   x8001      x3c25
       src/dest: 192.168.1.100 / 192.168.1.102
  Protocol: ICMP

OUTGOING PACKET (74 bytes):
  00 e0 18 97 62 e4 00 60   1d 02 0b 87 08 00 45 00   ....b..`......E.
  00 3c 68 09 00 00 3c 01   92 9d c0 a8 01 66 c0 a8   .<h...<......f..
  01 64 00 00 52 5c 02 00   01 00 61 62 63 64 65 66   .d..R\....abcdef
  67 68 69 6a 6b 6c 6d 6e   6f 70 71 72 73 74 75 76   ghijklmnopqrstuv
  77 61 62 63 64 65 66 67   68 69                     wabcdefghi
  Destination Host = 00:e0:18:97:62:e4
  Source Host =      00:60:1d:02:0b:87
  IP:  vhl/tos  len    id     offset  ttl/proto  csum
       x4500    x003c  x6809  x0000   x3c01      x929d
       src/dest: 192.168.1.102 / 192.168.1.100
  Protocol: ICMP
  Sent Echo Response
```

• **Listing 3: Verbose Ethernet Packet Trace**

Note that what you see in the Ethernet trace may vary because of other traffic that may be on your network. Be sure to go back to the uMON> prompt and type "ether on" one more time just to turn the verbosity back off.

2.6 The UDP-Based Command Interface

The serial port and network interface are now up and running, so the target is in good shape. This section will test/demonstrate MicroMonitor's ability to process commands over the network. Just as MicroMonitor's network stack is constantly monitoring incoming packets for ARP and ICMP, at the UDP level the stack is also monitoring the incoming packets for command requests[6]. This UDP based server, referred throughout this text as the moncmd server, simply takes an incoming UDP packet on port 777 and passes the content of the packet to the command interpreter. During the interval of time that the command is being processed, MicroMonitor's standard output is sent to both the console and the requesting client. For example, issue the following command at your host's console window[7]: moncmd 192.168.1.102 help. At the serial port of the target you should see…

```
uMON>MONCMD (from 192.168.1.100): help

Micro-Monitor Command Set:
arp         call        cast        cm          dhcp        dis
dm          echo        edit        ether       exit        flash
fm          gosub       goto        heap        help        ?
history     icmp        if          item        mt          mtrace
pm          prof        read        reg         reset       return
set         sleep       sm          strace      ulvl        tftp
tfs         unzip       xmodem      version
uMON>
```

Notice that the output of the "help" command was dumped to the serial port console of the target (plus the IP address of the client is shown) and also to the same console window on the PC that you executed moncmd. The string after the IP address is passed to the target as a MicroMonitor command. The string should be within double quotes so that white space (if any) is included in the message sent to the target. At the host console window, issue the command: moncmd 192.168.1.102 help dm . Notice that the output is the same, there was no "dm-specific" help output. This is because the moncmd tool only transfers the first argument after the IP address to the target, so the proper syntax for entering this command would be: moncmd 192.168.1.102 "help dm" .

Once again a the host console, issue the command moncmd 192.168.1.102 "@help dm" . Notice that the command was received and executed, but moncmd at the host eventually timed out. One relatively new feature in MicroMonitor's UDP command server is that it processes a leading '@' sign as a request to not send the output back to the client. The command is still processed, but the response is not sent to the client. In moncmd, there is an option to not wait for a response, so as one final demonstration of this facility, issue the command: moncmd –w0 192.168.1.102 "@help dm" . This time moncmd didn't wait or issue any kind of error message, it simply issued the command and terminated.

The moncmd server in MicroMonitor will also store the IP address and port of the most recent client interaction. The shell variable MONCMD_SRCIP will contain the IP address and MONCMD_SRCPORT will contain the port number.

An alternative to the umon-supplied moncmd tool is the "netcat" tool available with most Linux distributions. The command line "netcat -u 192.168.1.102 777" will put the user in an interactive mode that allows command entry and response to and from the target at IP address 192.168.1.102.

2.7 Wrap-Up

This chapter walked a new user with a new target system through the process of connecting a PC COM port to the target's serial port and verifying connectivity between the two. It briefly covered the items in the opening header as is displayed by MicroMonitor after a reset or power up. Since the first time use of target may require MAC address configuration, it also walked through that process. We configured and verified network connectivity, and demonstrated MicroMonitor's ability to trace Ethernet packets. Also, a few monitor commands

[6] When an application is in a mode "waiting" for some type of input from a network interface, the application is usually referred to as a "server".
[7] If you haven't already installed the MicroMonitor tools, refer to the uMon1.0 source distribution umon_main/host/README and do it now.

and concepts were used: shell variables, set, monrc, Ethernet trace, etc.. Quite a bit of stuff, not too much detail yet, but it's coming!

Chapter 3 Becoming Familiar with the Target

This chapter walks through all of the aspects of the MicroMonitor platform that can be demonstrated prior to actually adding an application. We've already established serial and network connectivity, so now we can talk about how to use each of these interfaces. Topics like the command line interface (CLI), the flash file system (TFS), transferring files and/or data using Xmodem and TFTP, etc.. This will all be done with a PC, HyperTerminal and a few other tools that come with the MicroMonitor package.

3.1 Getting Comfortable with the Command Line Interface (CLI)

All MicroMonitor commands are white-space delimited, case-sensitive strings of characters followed by ENTER. The first command to get comfortable with is "help". Type "help" followed by ENTER (hereafter the ENTER is assumed). All commands currently configured in the monitor will be displayed in a tabular format similar to the following…

```
uMON>help

Micro-Monitor Command Set:
arp         call        cast        cm          dhcp        dis
dm          echo        edit        ether       exit        flash
fm          gosub       goto        heap        help        ?
history     icmp        if          item        mt          mtrace
pm          prof        read        reg         reset       return
set         sleep       sm          strace      ulvl        tftp
tfs         unzip       xmodem      version
uMON>
```

• Listing 4: Tabular Output of the Help Command

To get a bit more information about each command, type "help –d". This will list each command with its required user level[8], plus a brief description of the purpose of the command...

```
Micro-Monitor Command Set:
 arp          0 Address resolution protocol
 call         0 Call embedded function
 cast         0 Cast a structure definition across data in memory.
 cm           0 Copy Memory
 dhcp         0 Issue a DHCP discover
 dis          0 Disassemble memory
 dm           0 Display Memory
 echo         0 Print string to local terminal
 edit         0 Edit file or buffer
 ether        0 Ethernet interface
 exit         0 Exit a script
 flash        0 Flash memory operations
 fm           0 Fill Memory
 gosub        0 Call a subroutine
 goto         0 Branch to file tag
 heap         0 Display heap statistics.
 help         0 Display command set
 ?            0 Display command set
 history      0 Display command history
 icmp         0 ICMP interface
 if           0 Conditional branching
 item         0 Extract an item from a list
 mt           0 Memory test
 mtrace       0 Configure/Dump memory trace.
 pm           0 Put to Memory
 read         0 Interactive shellvar entry
```

[8] More on "user levels" later.

```
reg          0 Display/modify content of monitor's register cache
reset        0 Reset monitor firmware
return       0 Return from subroutine
set          0 Shell variable operations
sleep        0 Second or msec delay (not precise)
sm           0 Search Memory
strace       0 Stack trace
ulvl         0 Display or modify current user level.
tftp         0 Trivial file transfer protocol
tfs          0 Tiny File System Interface
unzip        0 Decompress block of memory.
xmodem       0 Xmodem file transfer
version      0 Version information
```

• Listing 5: Help Output with Command Descriptions and User Level

Finally, for maximum detail on any single command type "help " followed by the name of the command. The output will contain at least three lines…

```
uMON> help dm
Display Memory
Usage: dm -[24bdefl:msv:] {addr} [byte-cnt]
Options:
 -2   short access
 -4   long access
 -b   binary
 -d   decimal
 -e   endian swap
 -f   fifo mode
 -l#  size of line (in bytes)
 -m   use 'more'
 -s   string
 -v {var} quietly load 'var' with element at addr

Required user level: 0
uMON>
```

• Listing 6: The DM Command's Help Output

Referring to the output of "help dm" (Listing 6), the first line is the same abstract that is shown with the "help –d" command (Listing 5). The second line is a summary of the usage of the command on the CLI. All following lines up to (but not including) the last line are lines specific to the command. This usually consists of a description of each of the options (if any) and possibly some additional notes on the command usage, or in some cases no text at all[9]. The final line is the required user level to execute the command.

When command usage is displayed, brackets [] and braces {} are used. The bracketed portions of the usage text indicate that the argument or option is not required; braces around an argument in the usage text indicate that the argument is required for the command to be successful. In the usage text above, notice that the "addr" argument is required, but the "byte-cnt" argument is optional. Also, note the summary of the options, -[24bdefl:msv:]. The brackets indicate that they are optional. The colon used after some of the option letters signifies that the option requires an argument. Following is an example of a valid "dm" command…

```
dm -2 -d -l8 0x100000
```

Breaking this command line down…

- **dm** is the command name.

[9] The INCLUDE_VERBOSEHELP configuration parameter in config.h controls whether or not this portion of the help text is included in the particular port of uMon. As a result, if your target is memory space sensitive, it is possible that this verbosity may not be present in the help output.

- **-2** is the option that tells dm to access memory as shorts (2-byte wide access).
- **-d** is the option that tells dm to print the output in decimal format
- **-l8** is the option that specifies the line size dm is to use. In this case we are specifying a line size of 8 characters, so '8' is the argument for the 'l' (the letter ell) option.
- **0x100000** is the required "addr" argument.

Notice that the "byte-cnt" argument is not specified, so a default is used. This is legal because the "byte-cnt" argument is in brackets; hence, not required. Also, the nature of the underlying function in MicroMonitor that supports command line options (getopt) allows options to be in random order and also allows options that do not require arguments to be concatenated with other options (omitting the dash), so the above command line could have been reduced to…

```
dm -2dl8 0x100000
```

Notice that if the options are kept separate, where each option is prefixed by a dash, then it is syntactically legal[10] for them to be in any order between the command name (i.e. "dm") and the first argument (i.e. 0x100000). So, the commands…

```
dm -2 -d -l8 0x100000
dm -l8 -d -2 0x100000
dm -l8 -d2 0x100000
dm -2d -l8 0x100000
```

are essentially identical. On the other hand, if the dash is omitted then care needs to be taken because although options that do not require an argument can be strung together in any order, if they are mixed with options that do require an argument (in this case, the "-l") , there can be confusion. For example, in the command…

```
dm -l82d 0x100000
```

the string "82d" would be processed as the argument to the "-l" option, so this is invalid syntax. The important thing to note here is that using the command line options provides a lot of flexibility within a given command (note the number of different options just for the "dm" command).

3.1.1 Command Line Editing and History

There are three possibilities regarding command line editing and history, depending on how the monitor was built: *vt100-arrows*, *readline-vi* style or no command line editing at all. You'll quickly discover which is installed by simply typing the up arrow key or ESC-k (escape key followed by the letter 'k'). These two different keystroke sets represent the "step back through CLI history" command for each mode. If the target reacts appropriately to one of them, then you know which one is installed. If neither seem to work, then it's possible that no command line editor was configured in when the monitor was built. The mode used on your target depends on how it was configured at build time.

VT100-mode: This is certainly the simpler and more intuitive of the two; however, it doesn't have as many features. After having entered the different versions of help above, you can now use the up-arrow key to retrieve them from the command line history[11]. The vertical arrow keys are used to scroll through command line history. The up arrow steps back through the history and the down arrow steps forward through the history. At any given point in typing at the CLI, the backspace key can be used to undo a character, plus the horizontal arrow keys can be used to shift left and right through the text without deleting a character. The CLI will be in "insert" mode; thus, using the left arrow to shift left a few characters, then typing a new character will cause that new character to be inserted into the string. To delete a character, use the arrow keys to move the cursor over (or before, depending on the cursor type) that character, then hit the delete key. At any point in this operation, ENTER can be used to cause the entire line to be passed into the monitor's command line processor.

[10] Note the term "syntactically legal". As the command executes it will always run the code associated with each option in the order it is processed on the command line; hence, in some cases there will be a functional difference when options are reordered.
[11] Your terminal emulator must be in VT100 emulation mode for the arrow keys to work properly.

Readline-vi-mode: This mode is a small subset of the vi-mode of command line editing that comes with most UNIX shells. "ESC-k" steps back through CLI history, and "ESC-j" steps forward. While typing a command, hit "ESC" to enter edit mode at any time. The following sequences are supported…

ESC k	step back through command history
ESC j	step forward through command history
h	move left on line
l	move right on line (character "ell")
0	move to beginning of line (character zero)
$	move to end of line
x	delete character
i	begin insert mode at current position
a	begin append mode at current position
R	begin replace mode at current position
r	replace single character at current position
D	delete from current position to end of line
dw	delete from current position to first whitespace
cw	replace word from current position to first whitespace

As you proceed through this document many of the commands in the MicroMonitor command set will be demonstrated, so you'll naturally get comfortable with the CLI as you progress.

3.1.2 Multiple Commands per Line with Looping

As of uMon 1.7, the command line supports the ability to put multiple commands on one line. Each command must be separated by a semicolon and the number of commands placed on one line is limited only by the size of the command line buffer (configurable via #define CMDLINESIZE, but defaulting to 256 bytes). Each command is processed just as it would be on a single line; thus, this simply provides the convenience of per-line command concatenation. The entire multi-command line is placed in the history buffer (described above), so the full line can be conveniently repeated.

Also, as part of this 1.7 enhancement, if the final character of the line is a left arrow, then the line is repeated in an endless loop. The loop is terminated by detecting a character input (similar to kbhit()) at the end of the command set of the line. Here are a few examples…

Example 1:
In this example, notice that the ADDR shell variable is modified by set and used by dm…

```
uMON>set ADDR 0x100; dm $ADDR 16; set ADDR=$ADDR+0x1000; dm $ADDR 16
00000100: ff 81 c6 02 ff 81 c6 0a   ff 81 c6 12 ff 81 c6 1a   ................
00001100: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
uMON>
```

Example 2:
In this example, the final character of the line is a left arror; hence, the line is repeated until the console notices an incoming character. In this case, after 7 iterations of the loop, I hit the spacebar to break out of it…

```
uMON>echo "hi \c"; echo there!<
hi there!
hi there!
hi there!
hi there!
hi there!
hi there!
hi there!
uMON>
```

3.2 The MicroMonitor Startup Environment

When the MicroMonitor firmware starts up, it has to do some basic initialization of the target system hardware. This includes configuring flash and RAM[12] accesses and configuring a serial and/or Ethernet port (usually both). Some of this configuration process creates information that is useful to the user, and some of it requires information from the user to properly complete. MicroMonitor maintains an "environment" to deal with this. An environment, in this context, simply refers to the monitor's ability to establish and deal with shell variables. This section introduces shell variables within MicroMonitor. It will discuss shell variables that are established by the monitor to provide accessible information for the user as well as those that can be set up by the user to configure the monitor as it is starting up.

3.2.1 The "set" command

The command used by MicroMonitor to maintain its environment is the "set" command. This command allows the user to modify, create and delete shell variables. Following is the output of "help set"…

```
uMON>help set
Shell variable operations
Usage: set -[acdef:imox] [varname] [value]
 -a        AND var with value
 -b        set console baud rate
 -c        clear the environment
 -d        decrease var by value (or 1)
 -e        build an environ string
 -f{file}  create script from environment
 -i        increase var by value (or 1)
 -o        OR var with value
 -x        result is hex (else decimal)

Required user level: 0
uMON>
```

As a review of the usage syntax described earlier, note that the options are not required, nor are either of the arguments (varname and value). This means that the "set" string alone is a valid MicroMonitor command…

```
uMON>set
       PROMPT = uMON>
   APPRAMBASE = 0xa0300000
  BOOTROMBASE = 0xbfc00000
     PLATFORM = Cogent CSB655
 MONITORBUILT = Jun 13 2005 @ 23:21:29
    MONCOMPTR = 0xa0000010
  VERSION_MAJ = 1
  VERSION_MIN = 0
  VERSION_TGT = 1
         ARG0 = monrc
         ARGC = 1
      CMDSTAT = PASS
        IPADD = 192.168.1.110
      NETMASK = 255.255.255.0
       GIPADD = 192.168.1.1
  CONSOLEBAUD = 38400
     ETHERADD = 00:23:31:55:00:11
uMON>
```

The above list is an example; your output may be slightly different. Generally speaking, this displays all of the shell variables (and their content) currently established in the MicroMonitor environment. Each variable was created by the monitor in one way or another, either intrinsically or through the automatic execution of the monrc file we created earlier. Later on we'll discuss in detail the command line syntax for using shell variables, plus the full list of shell variables used internally by MicroMonitor. For now, we just want to emphasize the few that are involved with basic startup of the monitor platform.

[12] In general, the term "RAM" is used to refer to the system's volatile memory, whether it be SRAM or DRAM.

3.2.2 CONSOLEBAUD

This variable contains the baud rate of the console. All MicroMonitor builds have some default, hard-coded baud rate that the console runs at. The console serial port is initialized very early in the bootup process so that it can be used to log error and/or status messages. After the monrc script file is run, the MicroMonitor firmware checks for the presence of the CONSOLEBAUD shell variable. If it is not found, then the firmware initializes the variable internally to contain the baud rate that the monitor was hard-coded to start up with. If, on the other hand, the shell variable is found (meaning that it was initialized in the monrc file), then this value is used to re-initialize the console baud rate. In either case, once the startup has completed, this variable will contain the value of the console baud rate; thus allowing an application that is run later to sync up to the baud rate of the monitor.

3.2.3 IPADD, GIPADD, NETMASK & ETHERADD

These four variables are used to initialize the Ethernet port. ETHERADD contains the target's MAC address. If this address is 00:00:00:00:00:00, this tells the MicroMonitor startup firmware to disable the Ethernet port. All other values are considered legal. IPADD, GIPADD and NETMASK are used to store the target's IP address (IPADD) , the gateway IP address (GIPADD) and network mask (NETMASK). All three are formatted as standard IP address "decimal dot" notation (i.e. 1.2.3.4). Recall from section 2.5 above, that these variables are typically stored in the monrc file so that they can be configured by the user, then at startup the MicroMonitor firmware will use them to configure the network connection.

3.2.4 APPRAMBASE

The monitor uses some RAM for its own internal .bss and stack space; however, the majority of the on-board RAM is dedicated to the application. For the monitor and an application to stay coordinated on the same target, there must be some way for the developer to be able to determine where the boundary is. The address stored in this shell variable is the beginning of RAM memory space that can be used by the application. The monitor internally considers this to be the start of memory space that is not allocated to the monitor's .bss/stack space. Several monitor utilities use the content of this address as a starting point for what the monitor considers to be "scratch" memory when no application is active. It's used as the default destination address for downloading with XMODEM and TFTP. It's the default base address for the monitor's built-in file editor. Most importantly though, it is the point at which the application can assume it owns memory once it is running. This address is used as a reference when determining what the base address of the application should be set to[13].

As a first example of actually using shell variables, invoke the command "pm" (put memory) to write some data to the memory at $APPRAMBASE, then the "dm" (display memory) command to verify that it is there… The CLI will convert shell variables to their values, so enter the commands:

```
uMON>pm $APPRAMBASE 0x31 0x32 0x33 0x34
uMON>dm $APPRAMBASE 4
0001c000: 31 32 33 34                                1234
```

The pm command writes four bytes to the address stored in the APPRAMBASE shell variable, then the dm command dumps those four values back to the console, in both hex and ASCII. Notice that this is safe because the monitor guarantees that the memory starting at this address is not used by the monitor itself.

3.3 The Startup File: monrc

Well, we've been eluding to it for the last few sections, so it's now time to talk more about it. For folks that use PCs there's the "autoexec.bat" file. For folks that use Unix, there's the ".profile" file. For users of MicroMonitor there's the "monrc" file. That pretty much says it all. The intent of the monrc file is to provide the target system with some mechanism to conveniently be able to modify the target's startup configuration, then have that configuration reload whenever the target restarts. It's an automatically invoked startup script for the target. Like its Unix and DOS counterparts, **you need to be careful with this file because you can't abort it**; hence, if you put an endless loop in the logic of the monrc script, you just killed your target startup.

[13] It is sometimes handy to leave a small amount of memory space between the address stored in APPRAMBASE and the actual starting point of the application. This provides a few benefits: it allows the monitor's bss/stack to grow without need to change the application's memory map; plus, some of the facilities within the monitor that use this space can still be used at application runtime.

We'll spend a lot of time working with files and scripts later. For now, just type the command "tfs ls" at the uMON> prompt. The output should indicate the presence of the monrc file that we created back in section 2.5 …

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 monrc                       73  0xff8c625c  e      envsetup

Total: 3 items listed (5608 bytes).
uMON>
```

Notice the lower case 'e' under the "Flags" column. This is indication that the file in TFS is an executable script. The monrc file must be created as an executable script. When we created it earlier, the "set –f" command was used, and it automatically created it with the 'e' flag set. Now type "tfs cat monrc" to show the content of the file…

```
uMON>tfs cat monrc
set IPADD 192.168.1.102
set NETMASK 255.255.255.0
set GIPADD 192.168.1.1
uMON>
```

When run, each line in the file is executed just as Unix would execute a script or DOS would execute a batch file. Typically, this file will contain the shell variable initializations needed for the Ethernet and console ports (sections 3.2.2 & 3.2.3 above). This accurately implies that the monrc file execution is done *during* MicroMonitor's startup sequence. Pretty clearly it is invoked prior to the initialization of the Ethernet port; the variables are established by monrc so that they can be used by the Ethernet driver initialization.

Generally speaking, this is the extent of the monrc file. **It is NOT to be used as the means of starting up the application automatically**[14]**.** As you progress through this text, you will see that there are other facilities within MicroMonitor's flash file system (TFS) that deal with that. The monrc file should be kept as simple as possible to avoid the hazard of accidentally putting some irreversible logic in that file and causing the target to not boot properly.

As of uMon1.0, MicroMonitor can be configured to allow the execution of the monrc file to be aborted, refer to the TFS_AUTOBOOT_ABORTABLE #define in section 11.10.1. Depending on the security needs of your system, this may be ok, and it does allow you to recover from a corrupt or errored startup. On the other hand, the fact that the monrc file is not abortable supports some level of security in your target startup. You can be assured that whatever you put in the monrc file WILL be done[15]. The primary value of this is best realized when used in conjunction with the monitor's user levels as discussed in section 5.6.

3.4 Using Shell Variables and Symbols

As I've already mentioned, the monitor supports the ability to assign data to or retrieve data from shell variables. The following text below discusses how shell variables and symbols are accessed on the command line, later in 0 each shell variable intrinsic to the monitor is discussed.

3.4.1 Shell Variable Usage

Similar to most other shells, shell variable names can contain alphanumeric characters and the underscore ('_'). Once assigned, the value within the variable can be accessed by preceding the variable name with a dollar sign ($). The ${} syntax is also supported. Use of the curly braces tells the command line processor to force the start and end point of a shell variable name that may otherwise not be seen as a variable because it is embedded within a larger string of characters. In addition, the dollar sign can be preceded by a backslash to negate its meaning as a shell variable starting delimiter.

[14] As of uMon 1.0, no binary exectuable can be launched out of monrc, and some commands are considered illegal (particularly those related to ethernet).

[15] Assuming no hardware intervention of course.

The MicroMonitor command interpreter supports nested shell variable names. This means that the user can build a shell variable name based on other shell variables. For example, if NAME_1 contains "Jane", NAME_2 contains "John" and IDX contains "1" then the statement echo ${NAME_${IDX}} will output the string "Jane"; likewise if IDX contains "2", then that same statement will output the string "John". This mechanism of processing nested shell variables within the command line interface allows the user to conveniently use shell variables to represent n-dimensional arrays, and by default, only the space used by the individual members within each dimension take up memory space. For example, assume we have an application that is represented by 3 entries in a table and the entries may exist in any of the first 100 locations of the table. When the entries are created they use the index into the table as part of their name: TBL3, TBL48, TBL99. Then the user can index into the table with a IDX variable... ${TBL${IDX}} (where IDX is set to 3, 48 or 99) and the content of each respective variable will be available. Note that although the application appears to have a table that contains at least 100 entries, the shell actually only assigns those that are used. A multi-dimensional array is handled the same way... PLOT[3][4][5] can be represented with the shell variable PLOT3_4_5 ... ${PLOT${X}_${Y}_${Z}}, where X=3, Y=4 and Z=5.

Refer to the discussion on scripts (Chapter 7 below) for additional examples of shell variable usage on the command line.

3.4.2 Shell Variables vs. Symbols

The command line processor makes token substitutions based on shell variable assignment and symbol table entries. As previously mentioned, shell variables are identified by preceding a string with a dollar sign ($). Symbol table entries are identified by preceding a string with a percent sign (%). The access to shell variables and symbols on the command line is essentially identical except for the '%' instead of '$''; however, the way in which they are stored (and the intent) is different; hence, either may be appropriate depending on the situation.

A shell variable is created on the command line of the monitor using the "set" command or through the API using the mon_setenv() function. Retrieval of the content on the command line is done by specifying the variable name preceded by a dollar sign ($) or through the API using the mon_getenv() function.

A symbol is not created; it exists or it doesn't exist based on the content of a symbol table file in the files system. Retrieval of the content of a symbol on the command line is done by specifying the symbol name preceded by a percent sign (%). Within an application, the monitor's mon_getsym() API function supports retrieval of a symbol. The default name for the symbol table file is "symtbl". At compile time, this can be overridden in config.h, or at runtime if the shell variable SYMFILE is set, then the content of that shell variable is used instead of the default "symtbl". The format of the symbol table file is simple: each line contains one symbol with a value that the symbol represents (each of which are whitespace delimited). For example, if the following lines were in the file "symtbl"...

```
main    0x14004
func    0x14800
```

then if %main was on the command line, it would be replaced with 0x14004; likewise if %func was on the command line, it would be replaced with 0x14800. This symbol-table querying ability allows the stand-alone monitor to provide symbolic querying of the application without the need for a host-resident debugger; plus, when it is no longer needed, the symbol file can simply be removed from TFS. As an example of symbol table substitution, the following command line

```
dm -4d %symname
```

would be converted to something like

```
dm -4d 0x401008
```

if the symbol table file existed and there was an entry for `symname`. Similarly for shell variables, the command line

```
arp $HOSTIP
```

would be converted to

```
arp 135.3.94.39
```

if the shell variable HOSTIP was previously set to 135.3.94.39.

The primary difference between shell variables and symbols is the way in which they are stored. Shell variables are allocated onto the monitor's heap so they use up RAM, symbol table entries are part of a file, so no RAM is involved[16].

NOTE1: The command interpreter can be told not to process the dollar sign by preceding it with the backslash (\). For example, assuming the shell variable $ABC is set to "hello", the command line "echo \$ABC = $ABC" could be used to output "$ABC = hello".

NOTE2: At the point in time when the command interpreter is making the substitution of the value of a variable in place of the variable name itself, if the variable does not exist, then no substitution is made. For example, if the shell variable ABC was previously set to "hello" then the command line "echo $ABC" would print out "hello"; but if the shell variable ABC was not previously set to anything, then the line "echo $ABC" would print out "$ABC". For scripting then, a comparison can be made to determine if a shell variable exists in the current monitor environment: if $ABC seq \$ABC goto VARS_DOESNT_EXIST.

NOTE3: Since the command line syntax of symbols and shell variables differs only in the use of '%' vs. '$', the logic of notes 1 & 2 above applies to symbols as well as variables.

3.5 Command Line Redirection

The monitor's command line supports the ability to log the output of a command to a memory buffer and ultimately to a file. At build time, the INCLUDE_REDIRECT macro must be set to 1 in config.h to enable this. At the command line, the syntax for doing this is similar, though not identical, to standard redirection on Unix. The difference is due to the fact that the redirection code within the monitor must be supplied with not only a file name, but also a buffer and buffer size. The idea here is that once supplied with a buffer, command output can be copied to this buffer and eventually the buffer will be transferred to a file. The syntax follows, with the redirection directives underlined...

```
uMON> echo this is some text >buffer,buffer_size[,filename]
```

This is the syntax for the single right arrow. A one or two comma delimited string containing a buffer address followed by the size of the buffer and an optional file name. If the filename is specified, then the output of the command is copied to the buffer (truncated at buffer_size if necessary) and then transferred to TFS as "filename". The running buffer pointer is reset back to the base address of buffer. If filename is omitted, then the output of the command is copied to the buffer and the pointer into the buffer is left at the position just after the data copied (assuming buffer_size is not reached).

```
uMON> echo this is more text >>[filename]
```

This syntax is used to append the output of the command to the buffer that was created by the '>' syntax described above. If 'filename' is present, then the content of the buffer is transferred to TFS as "filename" and the running buffer pointer is reset back to the base address of the buffer. If "filename" is not specified, there is no transfer to a file, the running pointer is incremented to the position just after the data copied (once again assuming buffer_size is not reached).

In both cases above, the "filename" string can contain up to 2 additional commas. These would be used to tell TFS the flags and info field to apply to the file when it is created, so the filename string could be "filename,flags,info".

[16] Future versions of MicroMonitor may support more complex symbol/shell variable retrieval by allowing a symbol to have members (a structure).

The memory space used is independent of the monitor's implementation of this capability; hence, it is the users responsibility to make sure that the specified buffer is memory space that can be used. There are a few different ways to determine a buffer area. Typically, the buffer would simply be $APPRAMBASE, but if the application is running or for some reason $APPRAMBASE is not appropriate, then the heap -m command can be used to allocate a buffer from the monitor's own heap space.

Here is an example with a buffer created and two additional command outputs appended to the buffer prior to it being transferred to a file (once again, the redirection directives are underlined)...

```
uMON> echo this is some text >$APPRAMBASE,400
uMON> echo this is another line of text >>
uMON> echo this is the final line >>logfile
```

Line 1 establishes the buffer to be 400 bytes starting at whatever address is stored in the APPRAMBASE shell variable[17]. The command output is copied to the starting point of the buffer. Line 2 appends the output of the second command to the buffer (no file transfer). Line 3 appends to the buffer and then transfers the buffer to "logfile". Execute the above three lines and then type “tfs cat logfile”. The content of the file is a duplicate of the output of the three lines…

```
uMON>tfs cat logfile
this is some text
this is another line of text
this is the final line
uMON>
```

Similarly, if access to the space at APPRAMBASE was not appropriate, then the monitor’s heap could be used. Following is a script that implements the logic…

```
 1: heap -m 400
 2: if $MALLOC eq 0 goto NO_SPACE
 3: echo this is some text >$MALLOC,400
 4: echo We're using memory from heap at $MALLOC >>
 5: echo this is more text >>
 6: echo this is end of text >>logfile1
 7: heap -f $MALLOC
 8: set MALLOC
 9: exit
10:
11: # NO_SPACE
12: set MALLOC
12: echo Could not be run due to low heap
13: exit
```

• Listing 7 : CLI Redirection Using Heap

This script shown in Listing 7 demonstrates a lot of different functionality. For the sake of this discussion, note the use of the heap command to allocate some memory that is accessible at the command line. Note that at all times during the buffer writes, if the running pointer reaches the end of the buffer (as specified by the buffer_size), then the logging is stopped. When, eventually, the directive is specified to write the file, it will have truncated to the size of the buffer.

3.6 Wrap-Up

We covered quite a bit in this chapter… Starting with the CLI, we discussed the command line editor and history as well as the format for all of the help output built into each MicroMonitor command. We’ve become familiar with how MicroMonitor starts up, establishing and expecting shell variables during startup. We mentioned a bit about MicroMonitor’s flash file system (TFS) and discussed the purpose and limitations of the “monrc” file.

[17] The use of shell variables to the right of the redirection arrow(s) is limited to the basic ‘$varname’ syntax, plus symbols are not applicable here.

Chapter 4 File/Data Transfer To and From the Target

For this chapter we will be talking about MicroMonitor and a few host based tools. MicroMonitor has the ability to transfer files to and from the target it resides on. MicroMonitor (the target-side of the transfer) allows the host to be unaware of the fact that the transfer may be to/from RAM or to/from a file in TFS. MicroMonitor comes with two standard mechanisms for transferring to and from the target. One mechanism uses the serial port (Xmodem) and one uses Ethernet (Trivial File Transfer Protocol or TFTP). Both of these protocols are simple lock-step protocols. Fairly easy to implement and can be found just about anywhere. On the host side of the transaction, HyperTerminal will provide the Xmodem interface, and ttftp (a tool that comes with the MicroMonitor package) will be used for the TFTP interface.

In all cases of data transferred to the target, the destination is initially to RAM. By default, the destination is the address that is stored in $APPRAMBASE, as discussed in section 3.2.4 above. After the transfer completes, if it is actually destined for a file in TFS, then the file is created from the data transferred to RAM. The following sections will walk through an example of each type of transfer, showing both the host dialog and MicroMonitor commands involved…

4.1 MicroMonitor's Xmodem Facility

When porting to a new target, this command is quite handy. It provides the ability to transfer files to/from the target prior to having the Ethernet interface running. It also supports the ability to download a binary image and automatically invoke the flash command sequence needed to reprogram the boot flash (like many things: dangerous but quite useful). We'll talk more about that later. Following is the output of "xmodem help" from the uMON> prompt…

```
uMON>help xmodem
Xmodem file transfer
Usage: xmodem -[a:BdF:f:i:s:uvy]
Options:
 -a{##}     address (overrides default of $APPRAMBASE)
 -B         boot sector reload
 -c         use crc (default = checksum)
 -d         download
 -F{name}   filename
 -f{flags}  file flags (see tfs)
 -i{info}   file info (see tfs)
 -s{##}     size (overrides computed size)
 -u         upload
 -v         verify only
 -y         use Ymodem extensions
Notes:
 * Either -d or -u must be specified (-B implies -d).
 * XMODEM forces a 128-byte modulo on file size.  The -s option
   can be used to override this when transferring a file to TFS.
 * File upload requires no address or size (size will be mod 128).
 * When using -B, it should be the ONLY command line option,
   it's purpose is to reprogram the boot sector, so be careful!

Required user level: 0
```

There are a lot of options to consider, most of which will be discussed in the command set description of Chapter 15. For now, the most important option is to specify the direction of the transfer, and if the default address stored in $APPRAMBASE is not appropriate, then the –a option should be used to tell Xmodem where to put the incoming data (assuming a download) or transfer the outgoing data from (assuming an upload).

With any Xmodem transfer, both the sender and receiver must be told to start up the protocol. In the following examples, the MicroMonitor side of the protocol is always started first, regardless of whether the target is the sender or receiver. MicroMonitor's Xmodem command waits for the other end to acknowledge the connection and then the transfer begins.

In preparation for the following examples, create the file C:\tmp\my_data_file.txt with the following ASCII text:

```
The rain in Spain falls mainly on the plain.
The wheels on the bus go round and round.
1234567890
SuperCalaFrajaListicExbeAllaDotious
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

Note that the data/filename is irrelevant; however, for the sake of staying coordinated with the text, using a file with the above name and data is recommended.

4.1.1 Xmodem: Data download

This is the simplest case. The goal is to transfer a block of data from a host system to the target somewhere in RAM. Obviously on the host system the data will be a file; however, on the target it will simply be some location in RAM. First, at the uMON> prompt, enter the command that will start the protocol up on the target side: "xmodem –d"[18]. Next, at the HyperTerminal menu click on the "Transfer->Send File…" menu item. Establish Xmodem as the protocol and click on the "Browse" button to choose a file to be transferred to the target. Enter the file name "C:\tmp\my_data_file.txt" that was created earlier. In this case we're allowing Xmodem to transfer the incoming data to the address already loaded in the $APPRAMBASE shell variable. If we wanted to use some other destination address, then we would just add "-a 0x123456" (where 0x123456 is the destination address that overrides the default).

Next click "Send" to begin transfer of the file to the target. The file is quite small, so you will probably see a very brief status box displayed by HyperTerminal, and then in the console window, MicroMonitor will indicate the number of packets it has received (should be 2)...

```
uMON>xmodem -d
□□□□□□□□□□□□□
Rcvd 2 pkts (256 bytes)
uMON>
```

Now type "dm $APPRAMBASE" and you will see that your file was transferred to that address…

```
uMON>dm $APPRAMBASE
0001c000: 54 68 65 20 72 61 69 6e   20 69 6e 20 53 70 61 69   The rain in Spai
0001c010: 6e 20 66 61 6c 6c 73 20   6d 61 69 6e 6c 79 20 6f   n falls mainly o
0001c020: 6e 20 74 68 65 20 70 6c   61 69 6e 2e 0d 0a 54 68   n the plain...Th
0001c030: 65 20 77 68 65 65 6c 73   20 6f 6e 20 74 68 65 20   e wheels on the
0001c040: 62 75 73 20 67 6f 20 72   6f 75 6e 64 20 61 6e 64   bus go round and
0001c050: 20 72 6f 75 6e 64 2e 0d   0a 31 32 33 34 35 36 37    round...1234567
0001c060: 38 39 30 0d 0a 53 75 70   65 72 43 61 6c 61 46 72   890..SuperCalaFr
0001c070: 61 6a 61 4c 69 73 74 69   63 45 78 62 65 41 6c 6c   ajaListicExbeAll
uMON>
```

The output of the above "dm" command shows the data in both ASCII-coded-hex as well as just straight ASCII. Note that "dm $APPRAMBASE" equates to "dm 0x1c000" as a result of the shell variable substitution[19].

4.1.2 Xmodem: Data upload

The upload interaction is similar to the download. Only difference is that now we're transferring data from the target to a host file. Using the previous section as the guide, specify "-u" instead of "-d" as the Xmodem command option to indicate that the transfer is an upload. In addition to the "-u" option, add "-s 166". This extra option is needed because the source of the transfer (in this case, MicroMonitor) must be told how

[18] Depending on the terminal emulator you use, you may see some noise after issuing this command. This is normal and can be ignored. This is the target-side of the protocol waiting for a connection with the host side.

[19] The actual address on your target system may be slightly different. It depends on the build time configuration of MicroMonitor on your target.

much data to send[20]. After entering this command at the uMON> prompt, click on the HyperTerminal menu item "Transfer->Receive File…". The name in HyperTerminal's dialog textbox is now a destination, and the interaction at this point depends on the version of HyperTerminal you have. You may enter a full filename or a path. If a path is requested, then after clicking the "Receive" button, enter a filename into which the transferred data will be placed. Either way, the goal is to transfer a block of 166 bytes from the target to a host file called C:\tmp\data_from_target.txt. So, take the appropriate steps to do that now.

```
uMON>xmodem -u -s166
Upload 256 bytes from 0x1c000
uMON>
```

Similar to the download steps above, you will see a brief dialog box displaying the transfer and at the uMON> prompt there will be indication that 256 bytes have been transferred. Notice from the above output, that the source address is once again 0x1c000 (content of APPRAMBASE); had we specified an alternative address using the –a option, this would have been different. If you open this file with NOTEPAD, you'll see that there is junk (random data) at the end of the file. This is due to the Xmodem protocol. The smallest block size used by Xmodem is 128 bytes, so extra junk is unfortunately appended to the file stored on the host.

4.1.3 Xmodem: File download

These next two transfers are similar, but now the target is dealing with a file instead of just raw memory space. At the uMON> prompt, enter the command: "xmodem –d –F my_file –s 166" (note the noise on the console once again). This command tells the target that the transfer is a download (-d) and that after the transfer is completely sent to the target RAM, it should be copied to the TFS file named "my_file". The –s option is used once again, to let the receiver of the file know what the file size should be; thus removing the "modulo-128" junk from the end of the file. Click on the HyperTerminal menu item "Transfer->Send File…" to once again bring up the dialog box. Use the same filename in the text box (C:\tmp\my_data_file.txt). Now click the "Send" button. Once again a quick status box will be seen from HyperTerminal and the output at the console will be…

```
uMON>xmodem -d -F my_file -s 166
□□□□□□□□□□□□□□□
Rcvd 2 pkts (256 bytes)
Writing to file 'my_file'...
uMON>
```

Upon completion of the transfer, issue the following commands at the uMON> prompt:

```
uMON>tfs ls
uMON>tfs cat my_file
```

The output of the "tfs ls" command shows:

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 monrc                       73  0xff8c625c  e      envsetup
 my_file                    166  0xff8c640c

Total: 2 items listed (239 bytes).
uMON>
```

This indicates that there is a new file in TFS. The size is 166 bytes and it is at flash address `0xff8c640c`. This address is likely to be different on your target; however, everything else (assuming you used the same file) should be the same. Note that the "Flags" and "Info" columns are empty for this new file. If the file was configured as an executable script, we could have specified that by adding a "–f e" option to the Xmodem command line, and the information string stored with the file could have been specified with the "-i" option[21]. The output of the "tfs cat my_file" shows:

[20] If we were transferring from a file, then this option would not be necessary because MicroMonitor would derive the size of the transfer from the size of the file.

[21] The "flags" and "info" fields are part of the file header. More on this when we get to TFS.

```
uMON>tfs cat my_file
The rain in Spain falls mainly on the plain.
The wheels on the bus go round and round.
1234567890
SuperCalaFrajaListicExbeAllaDotious
ABCDEFGHIJKLMNOPQRSTUVWXYZ
uMON>
```

This indicates that the file content is as expected.

4.1.4 Xmodem: File upload

The final example for Xmodem transfer is to upload a file from target to host. This is very similar to the previous examples, so we won't go into quite as much detail. The Xmodem command line is: "xmodem –u –F monrc". This indicates that we are going to upload the "monrc" file from the target to the host. Click on "Transfer->Receive File…" and, specify "C:\tmp\data_from_target1.txt" in the dialog. Upon completion of the transfer, the file "C:\tmp\data_from_target1.txt" will contain the same content as our target resident monrc file except that, once again, the Xmodem 128-byte block size will introduce some noise at the end of the file.

```
uMON>tfs cat monrc
set IPADD 192.168.1.102
set NETMASK 255.255.255.0
set GIPADD 192.168.1.1
uMON>
```

Notice in the above examples, when the target system is the receiver of the file, the Xmodem command in MicroMonitor can be told the exact file size; hence, there is no "modulo-128" junk at the end of the file. When the host is the receiver of the file, there's nothing in HyperTerminal that allows you to force the size of the file; hence, the junk is appended. The amount of junk depends on the size of the data (modulo 128). If the data size is 100 bytes, then there will be 28 bytes of junk. If the data size is 2 bytes, then there will be 126 bytes of junk, etc…

4.2 MicroMonitor's TFTP Facility

Now we start to use the much higher speed Ethernet port for data transfers. With this port available on the target system, it is very unlikely that the Xmodem method previously discussed will be used much. Nevertheless, for the porting process or for systems that don't have an Ethernet port, the Xmodem transfer mechanism is valuable.

We've already established network connectivity, and we have the necessary information to allow us to communicate with the target (recall that we already did this with the ping test of section 2.5). MicroMonitor supports both client and server modes of operation for TFTP. By default, a TFTP server is always running. The "tftp" command on the target is primarily used for the mode where the target is the client; however, it does support a few configuration parameters for the server.

```
uMON>help tftp
Trivial file transfer protocol
Usage: tftp -[aF:f:i:nvV] [on|off|IP] {get filename [addr]}
 -a        use netascii mode
 -F {file} name of tfs file to copy to
 -f {flgs} file flags (see tfs)
 -i {info} file info (see tfs)
 -v        low verbosity
 -V        high verbosity

Required user level: 0
uMON>
```

• Listing 8: TFTP Command's Help Output

To avoid overdoing it with detail, a similar set of examples (as was done with Xmodem) will be used to demonstrate the transfer process. We'll use the same files as we used for Xmodem, so to verify successful operation of the transfers, we start by purging the old stuff from both the host and target. On the host, remove

the file C:\tmp\data_from_target.txt. On the target we must remove the file "my_file" and also purge the RAM space of the previously downloaded data so that we can be sure that the new transfers actually worked.

The following command sequence at the MicroMonitor prompt displays the target content (commands are underlined)…

```
uMON>tfs ls
 Name                        Size   Location   Flags  Info
 monrc                         73  0xff8c625c  e      envsetup
 my_file                      166  0xff8c640c

Total: 2 items listed (239 bytes).
uMON>dm $APPRAMBASE
0001c000: 54 68 65 20 72 61 69 6e   20 69 6e 20 53 70 61 69   The rain in Spai
0001c010: 6e 20 66 61 6c 6c 73 20   6d 61 69 6e 6c 79 20 6f   n falls mainly o
0001c020: 6e 20 74 68 65 20 70 6c   61 69 6e 2e 0d 0a 54 68   n the plain...Th
0001c030: 65 20 77 68 65 65 6c 73   20 6f 6e 20 74 68 65 20   e wheels on the
0001c040: 62 75 73 20 67 6f 20 72   6f 75 6e 64 20 61 6e 64   bus go round and
0001c050: 20 72 6f 75 6e 64 2e 0d   0a 31 32 33 34 35 36 37    round...1234567
0001c060: 38 39 30 0d 0a 53 75 70   65 72 43 61 6c 61 46 72   890..SuperCalaFr
0001c070: 61 6a 61 4c 69 73 74 69   63 45 78 62 65 41 6c 6c   ajaListicExbeAll
uMON>
```

then removes the stuff…

```
uMON>tfs rm my_file
uMON>fm -c $APPRAMBASE 256 0
```

then demonstrates the fact that the data has been destroyed…

```
uMON>tfs ls
 Name                        Size   Location   Flags  Info
 monrc                         73  0xff8c625c  e      envsetup

Total: 1 item listed (73 bytes).
uMON>dm $APPRAMBASE
0001c000: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c010: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c020: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c030: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c040: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c050: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c060: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c070: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
```

4.2.1 TFTP Server: Data download

The TFTP server in MicroMonitor is always running by default; however, notice from the output of "help tftp" shown in Listing 8, that there is an "off" argument that allows the user to disable the server[22]. For these examples, we will leave the server enabled. With the target as the server, the host must be the client, so we will use the *ttftp* host-based tool that comes with the MicroMonitor package. Assuming you have ttftp.exe in your path, simply type "ttftp" at a DOS or BASH shell…

```
Usage:
 ttftp [options] {sysname} {get|put} {source} [destination]
 ttftp {srvr}

Options:
 -a     use netascii mode for 'put'; default is octet (binary)
 -q     suppress dots and ticker
```

[22] The use of the command "tftp off" is an example of a command that may be put in the monrc file to guarantee that the monitor comes up with the TFTP server disabled by default.

```
-t ##  inactivity timeout used by server (default is 60 seconds)
-T {Test_Setup_String} see below
-v     verbose mode, display details of each TFTP packet
-V     show version of ttftp.exe.

Test Setup String:
 -T supports the ability to inject delays, early termination and
 corruption into the stream of data fed to the client.
 Syntax...
 -T OPCODE,OPCODE_NUM,TESTTYPE,ARG1,ARG2
 Where...
  OPCODE is RRQ, WRQ, DAT, ACK or ERR;
  OPCODE_NUM specifies the 'nth' repetition of that opcode
  TESTTYPE is SLEEP, QUIT, CORRUPT
  ARG1 is SLEEPTIME, NA, or CORRUPTBYTE
  ARG2 is NA for now
 Examples...
 -T ACK,3,SLEEP,5
  On the third ACK received, sleep for 5 seconds, then continue;
 -T ACK,5,QUIT
  On the fifth ACK received, terminate;
```

• Listing 9: Usage Output of the Host-Based TTFTP Tool

The ttftp[23] tool includes facilities that are beyond the scope of this text, so simply ignore the "Test Setup String" descriptive text. Now enter the following command line at the host prompt:

```
ttftp 192.168.1.102 put C:/tmp/my_data_file.txt \$APPRAMBASE
```

• Listing 10: Complete TTFTP Command Line Example

Notice the backslash before the '$' in the line above. This is only necessary if the shell that you are issuing the command from will process the shell variable (BASH=yes, DOS=no). The backslash allows the '$' to be passed to the target, and this is what we want. Since the target is running the server, this ttftp client command simply starts up the transfer and the server processes it[24]. The client-server transactions complete and the end result is that the data is now on the target at the address that was stored in the $APPRAMBASE shell variable.

```
uMON>TFTP rcvd WRQ: file <$APPRAMBASE>
TFTP transfer complete.
```

• Listing 11: TFTP Response on Target

Note, in Listing 11, the server running on the target prints a few messages indicating that the transfer is in progress and that's it. Enter the MicroMonitor command: "dm $APPRAMBASE" to observe the fact that the data has been transferred…

```
uMON>dm $APPRAMBASE
0001c000: 54 68 65 20 72 61 69 6e   20 69 6e 20 53 70 61 69   The rain in Spai
0001c010: 6e 20 66 61 6c 6c 73 20   6d 61 69 6e 6c 79 20 6f   n falls mainly o
0001c020: 6e 20 74 68 65 20 70 6c   61 69 6e 2e 0d 0a 54 68   n the plain...Th
0001c030: 65 20 77 68 65 65 6c 73   20 6f 6e 20 74 68 65 20   e wheels on the
0001c040: 62 75 73 20 67 6f 20 72   6f 75 6e 64 20 61 6e 64   bus go round and
0001c050: 20 72 6f 75 6e 64 2e 0d   0a 31 32 33 34 35 36 37    round...1234567
0001c060: 38 39 30 0d 0a 53 75 70   65 72 43 61 6c 61 46 72   890..SuperCalaFr
0001c070: 61 6a 61 4c 69 73 74 69   63 45 78 62 65 41 6c 6c   ajaListicExbeAll
uMON>
```

[23] Yes, there are two 't's in ttftp. When I originally wrote this, it was called "Target TFTP".

[24] The "ttftp" tool will print one dot for each 512-byte block transfer made by the TFTP protocol. The files in this section are all small so in most cases only one dot will be displayed. Later in the text, larger files will be transferred with ttftp and a lot of dots will be printed.

• Listing 12: Memory Dump of $APPRAMBASE Address After TFTP Transfer

One important thing to notice here is that the destination file was the string "$APPRAMBASE". The TFTP client on the host sent that string to the TFTP server in MicroMonitor. Since MicroMonitor's TFTP server is running on an embedded system, it supports the ability to transfer to/from files or to/from raw memory space. In this case, the server detected that the incoming filename started with a '$', so it assumed it was a shell variable that needed to be dereferenced. Next, the server looked at the content of the shell variable and detected the leading "0x". This told the server that the location was an address rather than a filename[25]. So, in the above example, MicroMonitor's TFTP server converted the $APPRAMBASE string to the value of the variable (in this case 0x1c000), and treated it as an address rather than a filename because of the leading "0x".

4.2.2 TFTP Server: Data upload

A very similar set of transactions (as were used in the previous section) will be repeated to demonstrate this upload. The difference is that now the target is the source and the host PC is the destination. This time, enter the following command at the host prompt:

```
ttftp 192.168.1.102 get \$APPRAMBASE,166 C:/tmp/data_from_target.txt
```

Once again, the target-based server echoes the transaction. Notice that this time the server received an "RRQ" and on the previous transaction the server received a "WRQ". RRQ indicates a TFTP read request; hence the transfer is from the server. WRQ indicates a TFTP write request; hence the transfer was from to the server.

```
uMON>TFTP rcvd RRQ: file <$APPRAMBASE,166>
TFTP transfer complete.
```

Observe the content of the file C:\tmp\data_from_target.txt and notice that it is the same as the original file that we sent to that memory space earlier (C:\tmp\my_data_file.txt). Also notice that the source file name specified in the ttftp command line was "$APPRAMBASE,166". This is similar to the example of section 4.2.1, but now in addition to the $APPRAMBASE shell variable, there is a ",166" appended. The server deals with this comma-delimited syntax to support the ability to transfer some known size of raw data; otherwise, there would be no way to tell the server the size of the desired transfer because we weren't transferring a file of known size.[26]

4.2.3 TFTP Client: File download

Now we shift "TFTP gears" a bit. In the previous TFTP examples, the MicroMonitor based target was the server and the host pc (with ttftp.exe) was the client. To demonstrate MicroMonitor's TFTP client, this section will run with the opposite configuration: target is client, host machine is server. The ttftp tool can run as a simple server by invoking "ttftp srvr", so do that now. At the uMON> prompt, type in the following three commands:

```
tfs ls
tftp –F my_file 192.168.1.100 get C:\tmp\my_data_file.txt
tfs ls
```

Note that the IP address used in the above command should be whatever your host system's IP address is[27]. The result is that the MicroMonitor-based TFTP client retrieved the file from the PC-based TFTP server transferring the content of C:\tmp\my_data_file.txt on the host system into "my_file" in TFS. The initial output of "tfs ls" …

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 monrc                       73  0xff8c625c  e      envsetup

Total: 1 item listed (73 bytes).
uMON>
```

[25] Obviously this implies that MicroMonitor's TFTP server will not allow you to transfer a file whose name begins with "0x" or "$".
[26] This may at first seem a bit confusing; however, note that this syntax supports the ability to transfer random blocks of raw memory from the target to the host system.
[27] On the PC, use the command "ipconfig" to retrieve your host's IP address.

shows us that the only file in TFS at that point was the monrc file. The tftp client transaction…

```
uMON>tftp -F my_file 192.168.1.100 get C:\tmp\my_data_file.txt
Retrieving C:\tmp\my_data_file.txt from 192.168.1.100...
TFTP transfer complete.
Rcvd 166 bytes
Adding my_file (size=166) to TFS...
uMON>
```

then transferred the file from host to target, as can be seen by the final "tfs ls" command…

```
uMON>tfs ls
 Name                       Size   Location   Flags  Info
 monrc                        73  0xff8c625c  e      envsetup
 my_file                     166  0xff8c68dc

Total: 2 items listed (239 bytes).
uMON>
```

Also note that the transaction was logged in the console window of the PC…

```
TFTP transferring: C:/tmp/my_data_file.txt (octet)
TFTP C:/tmp/my_data_file.txt transfer complete
```

indicating that the TFTP server received a TFTP RRQ (read request) from the target and transferred the requested file.

4.2.4 TFTP Client: File upload

This is going to be a short one… MicroMonitor's TFTP client only supports "get"; hence, as a client, MicroMonitor can only retrieve files. All cases of data transfer are covered by MicroMonitor's TFTP server and the client's "get" capability.

4.2.5 MicroMonitor's TFTP Server and its Filename Convention

One final note on the topic of file transfers using MicroMonitor's TFTP server. We've mentioned the fact that files can be "autobootable" and "executable". These are attributes (or flags) associated with the file and if a file is to be transferred to TFS and is to contain one (or more) of these attributes, there must be some means of conveying that information in the download process. To deal with this, each flag is assigned a single-character (for example 'e' indicates executable script and 'B' indicates autobootable with query), and the complete filename is constructed with up to 2 comma delimiters. There are actually two additional fields that can be specified along with the filename: the flags and an optional information field. The generic syntax of a filename then is…

```
FILENAME[,FLAGS][,INFO]
```

So, to transfer a file named "myprog" with the flags 'e' and 'B', and the information field set to "sys_init", the syntax of the filename destined for TFS would be: "myprog,eB,sys_init". MicroMonitor's TFTP server looks for the comma delimiter and processes the incoming filename (with flags and info field) appropriately. So, as an example, if we wanted to transfer the "myprog" file with the attributes and information field as specified, the ttftp client command from the host would look like this:

```
ttftp 192.168.1.102 put C:/tmp/some_file myprog,eB,sys_init
```

And the resulting output of "tfs ls" (after the file was transferred) would look something like this:

```
uMON>tfs ls
 Name                       Size   Location   Flags  Info
 monrc                        73  0xff8c625c  e      envsetup
 myprog                      166  0xff8c68dc  Be     sys_init
```

```
Total: 2 items listed (239 bytes).
uMON>
```

In the case of a transfer of a file that does not have any flags, but does have an information field, the “FLAGS” portion of the syntax is omitted; however, the comma must be included. So, to transfer a file named “sysconfig” with only the information field set to “configuration”, the syntax of the filename destined for TFS would be: “sysconfig,,configuration”.

4.3 Wrap-Up

Xmodem and TFTP are both simple lock-step protocols. This simply means that when one side sends something to the other side, it doesn’t proceed until it receives some type of response or timeout. This type of protocol is simple and generally easy to implement, and is available from a variety of sources. Both mechanisms within MicroMonitor support the ability to transfer to/from raw or files. The Xmodem protocol is slow; however, it provides a means of transferring files without the need for an Ethernet driver. This makes it useful in the porting process as well as in systems that don’t have Ethernet. The only caveat with Xmodem (aside from the speed) is the 128-byte minimum block size; however this is actually only a problem on the receiving host side because on the MicroMonitor side, there are options to the Xmodem command that tell it where to chop off the last block prior to saving to a file. TFTP is clearly the better alternative when Ethernet is available. It’s faster, has no “modulo-128” issue and supports all of the same types of transfer as Xmodem; however, it does require a network. Bottom line… both transfer mechanisms have their place in embedded systems depending on the target situation.

Chapter 5 Application Startup

MicroMonitor's system startup can range from a single autobootable executable file installed in TFS, to a series of scripts that require an understanding of TFS, the scripting capabilities, shell variables and DHCP/BOOTP. As a result, the next two chapters will be touching a lot of topics. We'll look at the various attributes (or flags) that can be applied to files in TFS and see how they affect the way that MicroMonitor uses them. We'll discuss a lot more shell variables (especially when it comes to DHCP), some of which are read by MicroMonitor while others are established by MicroMonitor as a result of stepping through the DHCP/BOOTP protocol. Finally, we'll step through a few different examples of how MicroMonitor can be configured to startup a system:

- a configuration that simply has an autobootable application
- a configuration using a script to start up an application
- a configuration that uses the network to retrieve its IP, then boots an application
- a configuration that uses DHCP/BOOTP/TFTP to do a complete network-based bootup
- a configuration that boots off the network, but is also prepared to run standalone in the event that the network server is down.

5.1 TFS File Attributes and Info

We've mentioned various aspects of TFS quite a bit already; however, we really haven't elaborated much. Assuming you've read the previous chapters of this document and followed the steps outlined, then at this point you have a working serial interface and a working network interface. The serial interface comes up initialized automatically; however, the network interface requires you to establish a few shell variables prior to the initialization of the Ethernet driver. This led us to the creation of the special-case autobootable and executable *monrc* file (see section 3.3 above), which allows you to establish an environment for the target that is unique on the network[28]. So, how does MicroMonitor distinguish between "executable" and "non-executable" or between "autobootable" and "non-autobootable"?

TFS supports file attributes (or flags). The attribute simply describes the file to TFS, so that when TFS does some automatic stuff, it knows how to do it. An attribute in the file is simply a bit setting in the file header that lives with the file. At the command line, each attribute is assigned a letter which is used to display or create (in a non-verbose mode) the files in TFS. Take a look at the output of "tfs ls"…

```
uMON>tfs ls
 Name                        Size   Location   Flags  Info
 monrc                         73  0xff88a0dc  e      envsetup
 my_file                      166  0xff88a3ac

Total: 2 items listed (239 bytes).
uMON>
```

Note that "tfs –vv" [29] can be used to give a more verbose listing of each file…

```
uMON>tfs -vv ls
 Name:  'monrc'
 Info:  'envsetup'
 Flags: executable,
 Addr:  0xff88a0dc (hdr @ 0xff88a080, nxtptr = 0xff88a130)
 Size:  73 bytes (in sector 4)

 Name:  'my_file'
```

[28] Other MicroMonitor based targets on the same network would be configured with a different monrc file so that their network address information would also be unique.

[29] The tfs command's –v option has incremental values, meaning –v is verbosity level 1, -vv is level 2, etc… Refer to the manpage for the tfs command for more detail.

```
 Info:  ''
 Flags:
 Addr:  0xff88a3ac (hdr @ 0xff88a350, nxtptr = 0xff88a460)
 Size:  166 bytes (in sector 4)

Total: 2 accessible files (551 bytes).
uMON>
```

Each file has a name, size, location, flag and info field. Note, based on the above listing, that some files have flags assigned to them (as in the monrc file) and others do not (my_file). The fact that a file has no flag associated with it simply means that the file is just plain data. Following is a list of all the file attributes, including a brief description of each...

Attribute	Abbreviation	Description
executable script	e	a file of commands
auto-boot	b	file is to be run at boot time
auto-boot with query	B	file is to be run at boot time, after querying user
executable binary	E	COFF, ELF, AOUT, MSBIN, etc…
compressed	c	file is zlib compressed
in-place-modifiable	I	file is in-place-modifiable
unreadable	u	file not readable when monitor is below required user level
user_level	0-3	minimum user level required to access the file

5.2 Executable, Autobootable & monrc

Within TFS, the two main types of files are executable and non-executable. Just like most other computer systems, some files contain data and some files contain program.

5.2.1 Non-executables

Not too much can be said about non-executable files. They simply contain data. They can contain whatever is needed by the application being written for the target. The data can be stored in TFS and accessed through the TFS API by the application. TFS provides a very convenient mechanism for data storage on an embedded system. All data is stored as named files, making them location independent and accessible by name rather than address. The TFS API (discussed in more detail later) provides an easy-to-use API that supports the typical open/close/read/write model or allows you to directly copy a block of memory to a new file with one simple function call.

5.2.2 Executables

There are several different types of executable files that can be stored in TFS. Some of the file types are configured in to the monitor when it is built. At the highest level, TFS supports binary and ASCII executables. Similar to other computer systems, files can be loaded and run as native program code (binary) or interpreted line-by-line and run as a script or batch file (ASCII).

When MicroMonitor is built for a particular target, it is typically built to support one type of binary load file. As of this writing, MicroMonitor supports COFF, ELF, AOUT and MSBIN file formats[30]. In all cases, the load file is considered to be absolute; TFS does not do any relocation (although that may be a future enhancement). Most targets are built with the ELF format configured into TFS's loader[31]. This means that a standard ELF file, as it is generated by off-the-shelf GNU tools (and Microcross GNU X-Tools), can be transferred to the target's TFS space and run as is (assuming it's absolute memory map matches that of the target). Most of the supported loader file formats can also be compressed, since TFS supports zlib-based decompression.

[30] Additional formats can be added as long as there is a description of the file format available.
[31] Type "help tfs" to see which file formats are supported on your target.

Executables can also be "autobootable". There are three different types of autobootable files within TFS…

- Autoboot with query
  When the monitor starts up, this type of file tells TFS to query the user at the console to see if the automatic execution of the file should be aborted. If no interaction is detected within about 2 seconds, the autoboot continues; otherwise it is aborted.
- Autoboot without query
  When the monitor starts up, this type of file tells TFS to immediately autoboot the executable without allowing the user to intervene.
- Autoboot monrc
  This is a special case autobootable (the **mon**itor's **r**un-**c**ontrol file), that is automatically executed as a script prior to the monitor's Ethernet driver being initialized; hence, environment variables established in the monrc file are used by the driver initialization. **Do not use this file to automatically start up your application**. When the monrc script is automatically executed, MicroMonitor firmware is still initializing its own internals, so it isn't ready for an application at that point. Keep the monrc file simple, and it will serve you well!

Each of the above have their value. The monrc file is typically used to establish the network interface address and perhaps some other very basic environment setup. It cannot be aborted; thus, provides some guaranteed environment setup at startup. The ability to query the user is very handy during development because if the executable hangs the system, the query can be used to abort prior to the hang. The options give you ample flexibility, but you need to be cautious. Whenever you are dealing with a file that is autobootable and cannot be aborted at startup you need to be careful. The obvious case is that you may have a corrupt executable that will put the target in a state that is not recoverable; hence, you will need some kind of hardware intervention to restore the system. The less obvious case is that an application must contain some "hook" that allows it to gracefully return control to the monitor, or it must provide access to TFS for upgradeability. If not, then the application will run just fine, but it will not be upgradeable simply because there was no means to escape back to the monitor and there was no mechanism in the application to support upgrade.

5.3 The Application Script

Since this section discusses various ways to start up an application using MicroMonitor, we need some "dummy" application to use. If you're following this text in order, you'll note that we haven't actually built a real application yet, so now we'll build a small script that will simulate an application. Typically an application in an embedded system starts up and hopefully runs forever, or at least until some user intervention terminates it. Our script will do the same.

```
 1: #
 2: # my_first_app:
 3: # This script is used to simulate a real application
 4: # that can be started up through MicroMonitor.
 5:
 6: set COUNT 0
 7: echo Hello, this is the application!
 8:
 9: # APP_LOOP:
10: sleep 2
11: echo Loop $COUNT
12: set COUNT=$COUNT+1
13: if -t ngc goto APP_LOOP
14:
15: echo The application has been aborted!
16: echo Back to MicroMonitor...
```

• Listing 13: The "my_first_app" script.

A script in MicroMonitor is just like a script in UNIX or DOS. It's essentially a readable ASCII file that contains commands that can be part of the shell (i.e. built-ins), or may be programs written that are accessible from the PATH established in the environment. The only difference between this and MicroMonitor is that MicroMonitor doesn't have a directory hierarchy, so there is no need to worry about a PATH[32]. In the above example, each line is either a command or a comment. Commented lines start with a '#', and an empty line (or a line with only white space) is treated the same as a comment to provide readability. We haven't discussed all of the MicroMonitor commands yet, but hopefully the above set of commands look somewhat familiar, making the script somewhat intuitive[33]. The only tricky lines are 12 & 13. The expression evaluation capability of the "set" command (line12) provides the ability to perform an operation on a shell variable. In this case the content of the COUNT shell variable is being incremented by one. The "if" command (line 13) uses the –t ngc (test for not-gotachar) option to see if a character has been typed on the console.

So to summarize the above script, the first few lines are just comments, then it prints out a header to the console at startup ("Hello, this is the application!"), then sits in a loop printing "LOOP N" (where 'N' is the incrementing content of the COUNT shell variable) until the user hits a key at the console. At that point, the script terminates and returns control to the MicroMonitor platform that it was launched from. This will serve as an adequate simulation of a real application.

5.4 MicroMonitor's Startup Using Autobootable Files

At this point you have two files in your target's file system: a monrc file and a data file called "my_data", both of which were downloaded to the target during previous chapter's exercise on file transfer to/from the target. Next we want to transfer the above script file to TFS, and give it the "executable" attribute. Similar to section 4.2.3 above, the command to issue at your host shell is (use the appropriate IP address and source file path of course):

```
ttftp 192.168.1.102 put D:/files/my_first_app my_first_app,eB
```

Notice that the destination file in the command line above uses the comma-delimited syntax we mentioned in 4.2.5. The result of that command should be the addition of my_first_app to the file list[34]:

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 monrc                       73  0xff88a0dc  e      envsetup
 my_file                    166  0xff88a3ac
 my_first_app               334  0xff88a66c  Be

Total: 3 items listed (573 bytes).
uMON>
```

The file `my_first_app` is now installed in TFS and configured as an autobootable script (note the flags "Be"). If you type "my_first_app" at the uMON> prompt, MicroMonitor's CLI will recognize the fact that this is not a built-in command, it is an executable script and it will automatically execute it[35]. After a few iterations of "Loop N", hit the space bar to terminate the script…

```
uMON>my_first_app
Hello, this is the application!
Loop 0
Loop 1
Loop 2
Loop 3
```

[32] Just in case you're not familiar with it. the PATH shell variable is used in both UNIX and DOS as a way to find executable programs within a directory hierarchy of a computer system.

[33] Please note that it is beyond the scope of this section to go into all the details of MicroMonitor's scripting capability. That will be covered in a later chapter.

[34] Notice, that the files are listed in alphabetical order rather than in the order in which they were put in the file system.

[35] After a few passes through the loop, just hit a key on HyperTerminal to abort.

```
Loop 4
The application has been aborted!
Back to MicroMonitor...
uMON>
```

Hey, wait a minute! We just downloaded and ran our first MicroMonitor based application! Cool! Ok, it's not that cool, but its kind of neat if you're really a geek (I think its cool). Now, push the reset button on your board (or power cycle if there is no reset button). The output should look similar to this (once again after a few iterations of the "Loop N" hit the space bar)…

```
TFS Scanning //FLASH/...
MICRO MONITOR
CPU: MCF5272
Platform: Cogent CSB360 MCF5272 SBC
Built: Nov_15,2003 @ 17:30:33
Monitor RAM: 0x000400-0x01b84c
Application RAM Base: 0x01c000
MAC: 00:60:1d:02:0b:87
IP: 192.168.1.102
my_first_app?
Hello, this is the application!
Loop 0
Loop 1
Loop 2
Loop 3
The application has been aborted!
Back to MicroMonitor...
uMON>
```

Hopefully you noticed that there was a few second delay between the output of the "`my_first_app?`" line and "`Hello, this is the application!`". This is the "query" part of the autobootable file execution. Once MicroMonitor completes its own internal initialization (which includes invocation of monrc), it then scans through the list of files currently stored in TFS to see if any file is autobootable. If a file is autobootable, MicroMonitor then looks to see if it is an autoboot-with-query. If that is also true, MicroMonitor first displays the name of the file followed by a question mark to give the user the option to abort execution of that file. By default, the user gets about 2 seconds to abort this autoboot; otherwise the application will automatically startup (the actual delay can be set with the POLLTIMEOUT shell variable).

If you hit the reset button again and then hit the space bar (or any key) prior to the query timing out, you will abort the execution of the `my_first_app` script. The 'B' flag makes this file autobootable with query. If we re-downloaded the file again, but this time using a lower case 'b', MicroMonitor would not query the user prior to executing the file, it would immediately execute.

What if there is more than one autobootable file in TFS (not including monrc)? I mentioned earlier that the output of "tfs ls" lists the files in alphabetical order rather than the order in which they are actually stored in flash. TFS stores each file (and its header) in contiguous memory space, and each file is put back to back, basically creating a linked list in the flash. The files are not ordered alphabetically in the flash. As each new file is added, it is appended to the end of the linked list. As a result, the files can be in a variety of different orders physically in flash, but they will always be listed alphabetically. The same idea applies to the order in which autobootable files are executed (except for monrc, this file is always executed first if it exists and is flagged as executable). So, if TFS contained "my_first_app" and "script_a" as two autobootable files, the my_first_app file would be executed first and the script_a file would be next.

5.5 Autoboot Warning

The autoboot-with-query feature of TFS is quite useful. It allows you to download your application and run it, knowing that no matter what state the application puts the target in, you can recover from it by simply resetting the target and aborting at the time of the query. Great during development, but maybe not so great once a product is deployed. It's likely that you don't want your customer to have the option of aborting the

application startup[36], plus a few seconds of startup time are wasted because of the query time prior to running the application. The point is that you will probably want to shift over to autoboot-without-query. This is more secure; however, you need to make sure that your application is ready for this. There are two points that you must be aware of when transitioning to a non-query startup:

You need to be sure your application will start up successfully. If it autoboots and takes some kind of exception, that will just cause the target to restart and repeat that process in an endless loop. Without the ability to abort the startup[37], this then requires hardware intervention to clean up the flash and restore to a sane startup state.

Assuming you want to have the ability to upgrade your system in the future, then either your application has to have the ability to do this on its own, or it needs to have a simple hook that allows the user to gracefully have it return control to MicroMonitor. For example, in the above script, if the "if –t ngc" line wasn't there and you installed this script without the 'query' option of autoboot, then the board would reset and automatically run the script error free; however, you would have not ability to abort it and download a new script.

This is a very important point, and if your system is configured inconsistent with the design of your application, then you may not have a way to upgrade your application without some kind of hardware intervention. In the case of the "my_first_app" application, it was installed as autoboot-with-query; but even if it wasn't, it was designed to be aborted (by sending some character to the console once it is running). As a result, even if it were installed as autoboot-without-query, we would still have the ability to return control to the monitor.

5.6 Compressed Executables in TFS

Executables in TFS can be compressed in two different ways. The most obvious way is to simply use gzip on the host, transfer the file to TFS on the target, then use a script to unzip the content of the file to its destination location in RAM. The script would then be used to call some known entrypoint location with the decompressed area. The sequence of events would probably be as follows…

1. Create the exectuable ELF file on the host machine (say app.elf).
2. Convert that ELF file to a flat binary image (say app.elf.bin). This can be created with the host resident command: "`elf –B app.elf.bin app.elf`".
3. Compress that binary image (app.bin.gz).
4. Transfer app.bin.gz to the target.
5. Create a script that knows the load and entry points of the ELF file (say app.scr).
6. Transfer that to the target as an executable script (the 'e' flag must be set).
7. Now on the target, the script can be used to decompress the image and jump to its entrypoint.

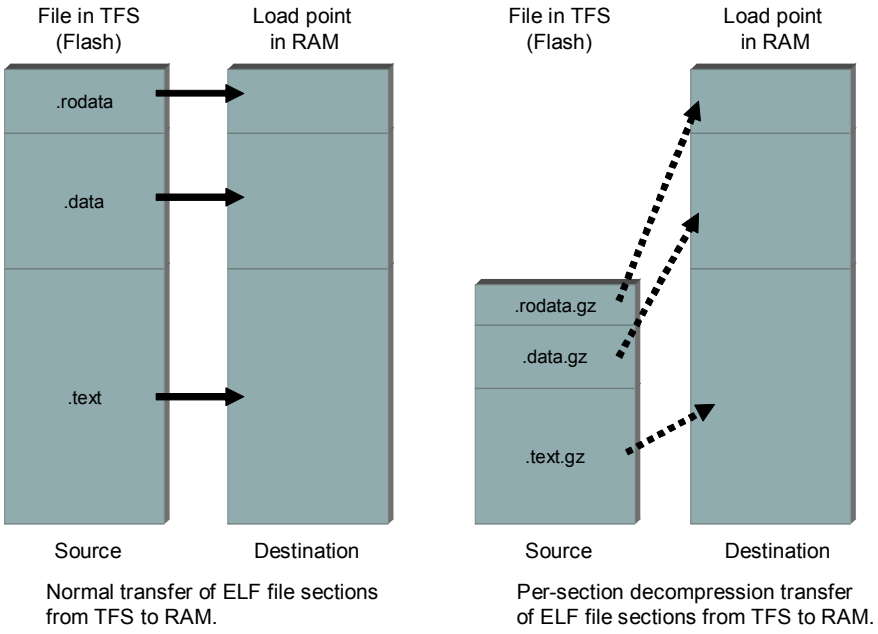
This works fine, but there are a few warnings that come with it… First, if the load or entrypoint of the executable ever changes, then the script must also change. Second, the bss area should be initialized by the script as well; hence, the script needs to know more about the memory map. Finally, if the memory map of the ELF file isn't back to back loadable sections, then memory is wasted on the .bin file because it contains both the loadable sections and the padding space between those sections. The padding must be stored in the .bin file simply because there is no intellient loader being used, so all the script can do is transfer one big file (padding space included) to memory.

TFS offers an alternative to this by providing the space efficiency that comes with compression plus the intelligence that comes with a real loader. ELF files can be stored in TFS as "compressed ELF". In this case, its not just a compression of the whole elf file to some app.elf.gz, rather it's an "ELF-aware" compression done on the host with the 'elf' tool. Each loadable section is pulled out of the ELF file, compressed and then placed back

[36] Not only does this abort the startup of your application, but it also gives the customer access to the monitor command line. Depending on the customer, this may or may not be acceptable. MicroMonitor's user levels may provide some help for this situation.

[37] The NO_EXCEPTION_RESTART shell variable can be set to TRUE (in monrc) so that if your application takes an exception it will not automatically restart. This, however, is something that would typically not be enabled in a field-ready system because obviously, if the target does take some kind of exception you want it to restart automatically.

into the ELF format. The end result is that the information for each section is still in tact, with the difference being that TFS not just copy the section from the file to RAM it will decompress the section from the file to RAM.



• **Figure 3: TFS ELF Decompression**

Referring to Figure 3, the left hand side of the diagram shows a standard transfer of an ELF file from file system to RAM. Each section is copied to some destination address in ram. The loader gets all of this information out of the ELF file format and the headers standard to that format. The right hand side of the diagram shows the improvement provided by TFS's per-section decompression. The file is still in ELF format, but each loadable section is now compressed; hence, a savings in flash space without losing the advantage of loading directly from and ELF formatted file. In this case, the sequence of events would be as follows…

1. Create the exectuable ELF file on the host machine (say app.elf).
2. Do the per-section file compression to create app.elf.ezip. This .ezip file is created by running the host-resident command: "`elf -z6 app.elf`".
3. Transfer app.elf.ezip to TFS and apply the 'Ec' flag to that file.
4. At the uMon command line simply type "app.elf.ezip" to load and run the application.

5.7 User Levels for Commands and Files

MicroMonitor supports the concept of user levels. The current implementation of the monitor supports 4 user levels (0 thru 3), with 3 being the highest user level (i.e. Superuser or Administrator). The files in TFS and the commands in the monitor shell can be configured to require a certain user level to gain access to the file or the ability to execute the command. In a nutshell, this means that various groups of files and commands can be restricted to only those users that know the password to get to the required user level (assuming a password file is installed).

At any given time, MicroMonitor is running at some user level. The *ulvl* command supports the ability to display the current user level as well as modify and/or configure the user levels. At startup, the target's default user level is at its highest (3) and all commands and files default to require user level 0 for access; hence, everything is accessible. If left untouched (via the ulvl command), the user level will remain at its max value and all facilities within the monitor will be accessible through the command line interface. If the running user level is lowered (by an autobootable script for example), then all accesses made to the hardware through

the monitor's user interface after that will be limited to the facilities (commands and files) that are available to the new user level. Commands and files can be configured to be accessible by some minimum user level. At system startup, all commands default to require that the system be at user level 0 or higher to execute (in other words, all commands can be executed). If commands are to be restricted to different user levels, then the *ulvl* command should be used in the monrc file to make all the necessary adjustments. For example, if the "flash" command is to be accessible only by user level 3, then in the monrc file (or some other autoboot-without-query script) the line "`ulvl -c flash,3`" will raise the required user level of the flash command and the line "`ulvl 0`" will lower the current user level of the system to 0; hence, to run the flash command, the user will need to know the user-level-3 password. Note that a default system startup does not have any password file installed, so access to the various user levels is approved with any password. The "`ulvl -p`" command must be used to create the three required passwords.

The usefulness of this feature depends on the needs of the application, but it basically provides the system with a mechanism to protect some portion of the system from unauthorized users. The user level can be raised, but only by a user that knows the password to get to that particular user level. The passwords are encrypted and stored in a file in TFS that is automatically saved at the highest user level. The file is also unreadable by user levels lower than the max. Password verification is done with one-way encryption by encrypting the entered password and comparing the result to the stored value. There is no decryption involved.

5.7.1 A User Level Example

Assume you have an embedded system project that is to be sold to a VAR (value-added-reseller). You want to use the same code base, but provide specific configuration parameters unique to each VAR you sell your box to. The VAR also wants to configure certain features based on the customer it is selling to, and finally, the end customer may have some administration parameters that it is able to configure on site. This situation calls for the ability to have certain parameters configurable by the manufacturer, but not modifiable (or possibly not even readable) by the VAR or its end user. Similarly, certain parameters are configurable by the VAR and should not be accessible by its customer. Finally, certain parameters are configurable by the customer, but should not be accessible by "guests" of the customers system. Each of these parameters should only be modifiable by the entity that is able to configure them; hence, the need for file access and command execution privilege levels.

This hierarchy of command and file access allows you to have certain portions of the system accessible only at user level 3 (those that you use to configure prior to shipment to the VAR). Then the VAR has certain configuration parameters set up at level 2, the customer has level 1 and, finally, a guest can log in at level 0. This allows a product to be shipped with a common code base that is configurable though files that are only accessible by the user level that needs to be able to access them.

By default, all commands are "visible" at all levels, but, since each command has a user level attached to it; each command will only be executable if that user level (or higher) is currently active. Likewise, for files, a file can have a user level attached to it and only be modifiable at that user level (or higher). Also, there is a "ulvl_unreadable" flag that can be associated with each file. This flag tells TFS that for all user levels below the file's designated level, the file is not even visible to the user; hence, files that contain passwords or other information that may be sensitive, can be "hidden" to lower user levels.

Since the idea of user levels requires some kind of password to put the monitor at that level, there is also a backdoor mechanism set up so that you can gain access to a system by knowing its MAC address. This essentially eliminates the possibility of losing accessibility because a password was lost. The backdoor password is an encrypted string based on the MAC address and can be created from a tool (maccrypt) that uses the same code as the monitor but runs on SUN Solaris and/or PC Win95/98/NT.

Obviously this is limited security, and if you really need to depend on it you need to make sure that you have properly assessed the "friendliness" of the customer or the individuals that will ultimately have access to the system. MicroMonitor provides this as a foundation that can be customized based on actual needs. The algorithms used to scramble the passwords stored in the TFS file are easily replaced by preferred in-house techniques that may be more appropriate, this includes the backdoor access mentioned above. Also, don't be fooled. Security is relative to the environment, and the desire of the individual to get in. This mechanism is quite

vulnerable to situations where the target's JTAG interface or address/data bus is physically accessible, and it is beyond the scope of MicroMonitor to provide security from that kind of attack.

5.7.2 The ulvl Command

First just execute the command "ulvl". Notice that the current user level is 3. This means you are essentially in superuser mode. Now, just in case you hadn't already noticed, type "help dm" and you'll see that there is a required user level for this command which, by default, is set to 0 (the lowest security level). All commands and files default to this level, and since MicroMonitor defaults to a running user level of 3, all commands and all files are accessible.

```
uMON>ulvl
Current monitor user level: 3
uMON>help ulvl
Display or modify current user level.
Usage: ulvl -[c:hp] [new_level|min|max] [password]
Options:
 -c{cmd,lvl}
      set command's user level
 -h   dump system header
 -p   build new password file

Note: with -c, if cmd is ALL, then all command levels are adjusted.

Required user level: 0
uMON>
```

The 'ulvl' command is the only command in MicroMonitor that cannot have its user level changed. It will always require only user level 0 to run. The command provides the following facilities:

The ability to adjust every other command's user level
The ability to adjust the current running user level
The ability to build a password file that contains the scrambled passwords for each of the user levels that requires one (1,2 & 3)
The ability to dump the system header, which allows the user to retrieve the target's MAC address for the backdoor discussed earlier.

We'll start by building a password file. Type the command "ulvl –p" and follow the interaction. Each password must be at least 8 characters. For this example set the following passwords:

Lvl1: userlevel1
Lvl2: userlevel2
Lvl3: userlevel3

Depending on the version of MicroMonitor you are using, the interaction may require you to enter the password and also verify by re-entering the password. Note that while entering the password, the characters are not echoed. After entering all three passwords, the command interaction will ask you to acknowledge acceptance of the configuration. Respond with 'y' to accept the new password file. This interaction creates a file in TFS; however, if you just type "tfs ls" you won't see the file. The filename is ".monpswd" and all files whose names start with a dot are invisible with a simple "tfs ls" command. To see this file, type "tfs –v ls"…

```
uMON>tfs -v ls
 Name                    Size   Location   Flags  Info
 .monpswd                  56  0xff8c725c  u3
 monrc                     73  0xff8c625c  e      envsetup
 my_file                  166  0xff8c68dc
 my_first_app             312  0xff8c70bc  Be
```

```
Total: 4 items listed (607 bytes).
uMON>
```

Notice the presence of the .monpswd file. More importantly, notice the flag associated with it is "u3". The 'u' indicates that the file is **u**nreadable below the specified user level. This has two affects:

When running at a user level less than three, the file will not be readable; hence the command "tfs cat .monpswd" will not dump the content of the file.

When running at a user level less than three, the file will not be listable; hence the command "tfs ls .monpswd" or "tfs –v ls" will not indicate the presence of the file. Note that this is not at all related to the invisibility provided by the "dot" prefix.

Try this… Type "tfs cat .monpswd". You should see three lines, each containing a string. Each of these is the one-way encrypted string that represents the password for a user level. Now, bring your user level down to 2 with the command: "ulvl 2". The command will respond by indicating your new user level of 2. Note that you don't need to enter a password for this because you are moving to user level 2 from a higher user level (3). Now type "tfs –v ls" and "tfs cat .monpswd" again. The listing doesn't show the file and the "cat" command doesn't allow access to the file's contents. The only way you can access this file is if you were at user level 3. To get back to user level three, enter the command "ulvl 3", and note that now you need to know the password that you configured when you ran the "ulvl –p" command earlier. Once you get your user level back to three you will be able to access the .monpswd file.

The same idea applies to commands in MicroMonitor's shell. By default all commands require a user level of at least 0 (making all commands accessible, since user level 0 is the lowest user level). Configure the "dm" command to require a minimum of user-level 1 with the command: "ulvl –cdm,1". The –c option of the ulvl command expects a comma-delimited string as its argument. The string before the comma is the command name and the string after the comma is the minimum user level that command should require. Now, lower your user level to zero by typing "ulvl 0", then type "help dm" and "dm $APPRAMBASE". Notice that the command is no longer accessible by help, and any invocation of the "dm" command not be completed.

So, what does this give you? Actually, as described, it has no value (yet). A user can simply reset the target and it will start up at user level 3 and all commands will be back to requiring user level 0. This is one good example where "monrc" or some other file configured as "autoboot-without-query" becomes useful. With the ability to run a script that cannot be aborted, the script can contain all of the "ulvl" commands that configure your system to run with the permissions that are applicable to your project. Initially, you can work out the details of what commands need to be at what user level and not have a password file installed. Each time a transition to a user level that would normally require a password occurs, the command will let you know that you are being granted access because of the lack of a .monpswd file. Then, once you've settled on your configuration, you can add the password file. Just don't forget the passwords!

The backdoor provides an escape from the situation where you've forgotten your passwords; however it also provides vulnerability. As the name implies, it provides an alternative route to get in to the system. To make each target unique, it uses the MAC address of the target to derive a password that will work at all user levels. This works with the host based maccrypt tool. The tool simply takes the MAC address as an argument and spits out a password string. Assuming the scrambling algorithms on the target and in the maccrypt tool are the same, then this provides a means to get into a system without knowing any of the three passwords. Good news: you don't lose your system if you forget the password. Bad news: only one password is needed to get access to all user levels. The MicroMonitor source can be configured to enable or disable this backdoor mechanism.

.

5.8 Wrap-Up

First of all, to avoid any possibility of forgetting the passwords, now is a good time to remove the .monpswd file from your target. This chapter introduced the first actual application run out of MicroMonitor. Granted it was a very simple example; nevertheless, it demonstrated the facilities that MicroMonitor provides the application for startup. We also covered some of the design steps that must be taken either by the way the application is configured in TFS (via flags) or by the capability built in to the application. One way or another

there needs to be some way to be able to update your system, so its important that the application does not cripple MicroMonitor's built-in mechanisms unless it is providing a mechanism of its own.

Finally, we discussed MicroMonitor's concept of user levels. Being able to specify commands and files to be accessible only at some user level (or above) allows the system to be deployed with a full set of commands and files, but then limit the accessibility of those commands and files through the user level. Lots of good stuff. Next chapter will dig into network bootup.

Chapter 6 Booting off the Network

Previously we've talked about the monrc file and how it can establish a "personality" for the target running MicroMonitor. Everything else on the target (except for the Ethernet MAC address) may be the same but with a few different settings in the monrc file you end up with multiple unique targets. In a networked environment the most obvious distinction between systems is the target's IP address. The monrc file certainly solves this problem, and it does it without the need to interact with any other device on the network.

In some cases, it's reasonable to assume that the target will be attached to a network whenever it boots. Within this context, there are several approaches that can be implemented thanks to the scripting capabilities of MicroMonitor. The target can be configured to do everything through DHCP or BOOTP; thus, requiring that the network be present, or it can be set up to attempt a network boot, but if the network boot fails, it can fall back to a local boot. This chapter will start with details on MicroMonitor's DHCP/BOOTP capabilities, then give an example of the standard network boot strategies, plus a few others that add flexibility in the cases where the presence of a reliable server is questionable. In all cases we will be using the "dhcpsrvr" tool (part of the uMon1.0 distribution) as the network side of the bootup.

6.1 DHCP/BOOTP Details:

A MicroMonitor based target can be configured to issue a BOOTP/DHCP broadcast. Shell variables are used to configure the outgoing request and to store some of the returned parameters from the server. The "dhcp" command supports startup of a DHCP or BOOTP client transaction. The '-b' option tells the command to use BOOTP, else DHCP is used. The only other options for this command are 2 verbosity levels –v to dump the IP settings upon completion of the handshake, and –V for a dump of all network packets involved in the transaction…

```
uMON>help dhcp
Issue a DHCP discover
Usage: dhcp -[bvV] [vsa]
Options...
 -b      use bootp
 -v|V    verbosity

Required user level: 0
```

DHCP & BOOTP are protocols that a network node can use to request startup information of some kind from a server assumed to be within reach of the subnet on which the requesting computer resides. In a nutshell (for complete details refer to RFCs), it's a 4-step handshake. The client issues a DISCOVER, server responds with a OFFER, client issues a REQUEST and server responds with a ACK.

- **DISCOVER**: request by the client broadcasting the fact that it is looking for a DHCP/BOOTP server.
- **OFFER**: reply from the server when it receives a DISCOVER request from a client. The offer may contain all the information that the DHCP client needs to bootup, but this is dependent on the configuration of the server.
- **REQUEST**: request by the client for the server (now known because an OFFER was received) to send it the information it needs.
- **ACK**: reply from the server with the information requested.

The 4-step handshake is used because of the possibility that the client will be on a network that has more than one DHCP/BOOTP server. On the initial broadcast of the DISCOVER, several servers may respond with an OFFER, and at that point the client must look at the offer and decide whether to accept it (and reply with server-specific REQUEST) or ignore it. Upon reception of the REQUEST, the server completes the transaction with an ACK, and within that ACK message is all of the startup information needed by the client issuing the request. Following are the DHCP and BOOTP header formats including what entries in the received header are transferred to shell variables in the monitor...

op (1)	htype (1)	hlen (1)	hops (1)
xid (4)			
secs (2)		**unused (2)**	
ciaddr (4)			
yiaddr (4) Copied to IPADD shell variable			
siaddr (4) Copied to BOOTSRVR shell variable			
giaddr (4) Copied to RLYAGNT shell variable			
chaddr (16)			
sname (64)			
file (128) Copied to BOOTFILE shell variable			
vend (64) Option #1 copied to NETMASK Option #3 copied to GIPADD Entire field copied to DHCPVSA			

• **Table 1: Structure of a BOOTP Packet**

op (1)	htype (1)	hlen (1)	hops (1)
xid (4)			
secs (2)		**flags (2)**	
ciaddr (4)			
yiaddr (4) Copied to IPADD shell variable			
siaddr (4) Copied to BOOTSRVR shell variable			
giaddr (4) Copied to RLYAGNT shell variable			
chaddr (16)			
sname (64)			
file (128) Copied to BOOTFILE shell variable			
options (312) Option #1 copied to NETMASK Option #3 copied to GIPADD Option #51 copied to DHCPLEASETIME Entire field copied to DHCPVSA			

• **Table 2: Structure of a DHCP Packet**

Referring to Table 1, upon completion of the BOOTP/DHCP handshake with some server, the target expects the *yiaddr* field to be loaded with the target's IP address. This value will be stored in the IPADD shell variable. In addition, it will store the content of the *siaddr* field into the BOOTSRVR shell variable, the *file* field into the BOOTFILE shell variable, and the *giaddr* field into the RLYAGNT shell variable. Certain standard options are also transferred to shell variables...the router (option code 3) into the GIPADD shell variable and the subnet-mask (option code 1) will be stored into the NETMASK shell variable. Note that each of these additional fields will be loaded into their respective shell variable only if found to be non-zero. Note also that if after the DHCP/BOOTP transaction has completed, the RLYAGNT variable is loaded, but GIPADD is not, then the content of RLYAGNT is copied to GIPADD. Finally, to satisfy other non-standard scenarios, the entire content of BOOTP vendor-specific-area of the DHCP options area will be copied (in ASCII-coded-hex) to the DHCPVSA shell variable if the DHCPVSA shell variable is present (to indicate that it should be overwritten with the field).

Upon completion of the BOOTP/DHCP handshake, if both the BOOTFILE and BOOTSRVR variables have been loaded, the target will attempt a TFTP transfer of the specified file from the specified server. Since the file (destined to reside in TFS) can be one of several different types, the target will look for the comma-delimited extension on the filename to be used as the file's TFS flags (discussed in section 4.2.5). For example, if the BOOTFILE variable contains “abcde,eB”, then the file will be stored into TFS with the name "abcde" and flags "eB". This filename extension will be processed as TFS flags only if each of the characters in that extension are legitimate TFS flags. As a final step in the BOOTP/DHCP startup, if the file is successfully transferred from the TFTP server, the target will then execute it as a standard executable file under TFS. Note that execution is attempted, but will only be carried out if the flags have been properly.

Note1: The incoming file is transferred to TFS using the API function tfsadd(). This means that if the file already exists in TFS and is 100% identical, there is no flash-write overhead.

Note2: If there are no specified flags for the file that is downloaded, then it is assumed to be 'e', meaning that the file is treated as an executable script.

WARNING: If the DHCP/BOOTP transaction is started up as a result of the IPADD shell variable being set to DHCP or BOOTP, then the handshake is done in the background while the monitor is booting up; hence, any autobootable scripts can run while the handshake is in progress. The user must be careful here... Potentially, an autobootable script and the handshake will be occurring at the same time, so the script must be aware of this. The important thing to note is that if the file that is downloaded into TFS is autobootable, then it will run while the BOOTP/DHCP handshake is potentially loading in a new version of the same file. This will definitely confuse things; so ideally, if the file loaded in by BOOTP/DHCP is executable, and then there should be no autobootable files in TFS. On the other hand, if the file loaded in is not executable, then some other autobootable file can be set up to be waiting for this to be loaded. Examples later in this chapter will demonstrate this.

6.2 DHCP Specifically...

DHCP and BOOTP are very similar. DHCP offers more flexibility. For DHCP, the monitor supports the 'automatic allocation' portion of the specification. There is no lease expiration, the IP address assigned at the time of the initial handshake is assumed to be owned by the target until it is reset. Basically this means that the DHCP supported by the monitor is an extension of BOOTP, providing a bit more flexibility with regard to the parameters that can be retrieved from the DHCP server and the way in which the client and server agree to handshake. The target issues a DHCP_DISCOVER broadcast. In that message, the flags, class identifier, client identifier and DHCP parameter-request-list may be loaded from the shell variables DHCPFLAGS, DHCPCLASSID , DHCPCLIENTID and DHCPRQSTLIST respectively (refer to 0 below). The DHCP server may be equipped to expect that and based on its configuration, the server may respond to the broadcast with a DHCP_OFFER. By default, the offer is accepted, but if the DHCPOFFRFLTR shell variable (see section 14.18) is set, some filtering is done to determine whether or not the client should proceed or just ignore the offer. If the client proceeds, then the target issues the DHCP_REQUEST and the server replies with a DHCP_ACK. The payload accompanying this final acknowledgement is what contains all of the target-specific parameters from DHCP.

As mentioned above, the monitor does not support lease expiration; however, it does provide hooks to allow an overlaying OS to handle lease renewal. By default, the incoming lease time option is ignored. The client accepts the server's offer and assumes infinite lease time. If, in the monrc file, the shell variable DHCPLEASETIME is set, then the content of this variable is used as a minimum that the incoming lease time from the server is compared to. If the incoming lease time is greater than or equal to this value, then the offer is accepted and the DHCPLEASETIME shell variable is reloaded with the incoming lease time. If there is no incoming lease time specified by the server, then the DHCPLEASETIME shell variable is cleared. The OS can then look at this shell variable (if present) to see when to issue a lease renewal (if at all).

6.3 Preparing the Server

This section is only necessary if you plan to use the DHCP/BOOTP server that comes with the MicroMonitor package (dhcpsrvr). If you have your own, or you just plan to use one that is already running on your network, then you can skip this section. However, note that in the following sections there will be some reference to this section regarding how the various pieces of information were transferred from server to client.

The dhcpsrvr tool is a basic DHCP/BOOTP/TFTP server that can be used for developing and/or testing an embedded client. It is not intended that this server be used to ever deal with more than one client at a time. It can be configured to respond to a client based on the client's MAC address. The tool depends on a dhcpsrvr.cfg file for establishing what clients it responds to as well as what is included in the response. The tool can be used to create a default config file that contains a verbose example of what a real config file should look like. The command line "dhcpsrvr –C" will dump a default configuration file to standard out, so redirecting that to a file called "dhcpsrvr.cfg" and then editing that file is all that is needed to build a configuration file. Complete details of the command can be found in section 17.8 below. For now, we'll just establish a basic configuration.

From the output of dhcpsrvr –C, create the dhcpsrvr.cfg file shown in Listing 14. For each of the entries, the actual MAC and IP addresses used will depend on your network, so it is very unlikely that you will use this exact example. Modify as needed.

```
BOOTP_CLIENT_MAC: 00:60:1D:02:0B:87
CLIENT_IP:        192.168.1.102
SERVER_IP:        192.168.1.100
RLYAGNT_IP:       192.168.1.1
NETMASK:          255.255.255.0
GATEWAY:          192.168.1.1
BOOTFILE:         my_first_app
```

- **Listing 14: Simple DHCP Server Config File (dhcpsrvr.cfg)**

WARNING: Prior to starting up the server, it's best if you isolate yourself from other devices on your network. The activity generated by the transaction between the target system and dhcpsrvr tool may cause confusion on your network. Plus, to reduce confusion during this discussion, it is best that only one DHCP server reply to the request generated by MicroMonitor.

6.4 A BOOTP Example

To best see what the result of the transaction is, remove the environment and some files from the target system. At the uMON> prompt, issue the command "tfs rm FNAME" (where FNAME is a filename) for each file listed by "tfs ls" except for monrc. Next, issue the command "set –c" at the uMON> prompt so that all shell variables are cleared from the current environment. At this point the file system and environment are "clean". The output of "set" and "tfs ls" should be similar to Listing 15…

```
uMON>set
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 monrc                       73  0xff88005c  e

Total: 1 item listed (73 bytes).
uMON>
```

- **Listing 15: Environment and File Dump Prior to BOOTP Transaction**

Note an empty response from the "set" command and a single file in TFS flash space. Now at a console window of the PC, run the dhcpsrvr command with a –T (startup a TFTP server also) option in the same directory as the dhcpsrvr.cfg file was installed. It will immediately return a message indicating that the server is running. Also in that directory place a copy of the my_first_app script that we built earlier. This first example will demonstrate how BOOTP can be used to retrieve basic network information as well as a boot-up file that can be used to start up the system. With the server running on the PC, now just type "dhcp –b" at the uMON> prompt.

```
INCOMING BOOTP:  (mac = 00:60:1d:02:0b:87)
OUTGOING BOOTP:  (mac = 00:60:1d:02:0b:87, ip = 192.168.1.102)
TFTP transferring: my_first_app (octet)
TFTP my_first_app transfer complete
```

• **Listing 16: BOOTP Trace**

```
uMON>dhcp -b
uMON>Retrieving my_first_app from 192.168.1.100...TFTP transfer complete.
334 bytes
Adding my_first_app (size=334) to TFS...
Hello, this is the application!
Loop 0
Loop 1
Loop 2
```

• **Listing 17: BOOTP File Retrieval and Automatic Execution**

At the PC, the server should have dumped something similar to Listing 16 and at the target, the output should be similar to Listing 17 (at the target, just type any character to abort the loop). The sequence of events are as follows:

The server (DHCP/BOOTP and TFTP) was started up on the PC.
The client issued the bootp request (the dhcp –b command at the uMON> prompt).
The server received the request, verified that the MAC address of the request was in its configuration, and replied with the information specific to that MAC address in the dhcpsrvr.cfg file.
Upon receiving the response from the server, the client established its IP address and used the presence of the BOOTFILE and SERVER_IP entries in the BOOTP response to generate a TFTP request for the file "my_first_app" to the server at IP address 192.168.1.100.
The server received the request and transferred the file to the target.
The file was stored in TFS and run as an executable script.

That was a complete BOOTP transaction. The previously "empty" target now has the file "my_first_app" plus an environment established as a result of one BOOTP transaction. Once again type "tfs ls" and "set" to see the results.

```
uMON>tfs ls
 Name                       Size   Location   Flags  Info
 monrc                        73  0xff88005c  e
 my_first_app                334  0xff8802bc  e

Total: 2 items listed (407 bytes).
uMON>set
        BOOTFILE = my_first_app
        BOOTSRVR = 192.168.1.100
           IPADD = 192.168.1.102
         TFTPGET = 334
           COUNT = 3
uMON>
```

• **Listing 18: Environment and File Dump After the BOOTP Transaction**

The actual value of COUNT will depend on when you halted the loop, but all the other information in Listing 18 should be very similar to your output. Note the presence of the TFTPGET variable. This is a result of the TFTP transfer to allow a script to be aware of the success of the transfer. In MicroMonitor, there are three different ways to turn this target into a real network-booted system:

In the monrc file, set the IPADD shell variable to BOOTP (or, for DHCP, set it to DHCP)
A simple autobootable script could be put on the target with the line "dhcp –b".
MicroMonitor can be built with a hard-coded BOOTP (or DHCP) as the default startup mechanism for cases where even a monrc file does not exist. In the config.h file, the value of DEFAULT_IPADD can be set to the string "BOOTP" (or "DHCP"). This could then be overridden by a monrc file, but provides a mechanism by which a system can be configured in the event of a non-present monrc file.

Any one of these mechanisms would configure the target to automatically startup the BOOTP (or DHCP) transaction after a reset. In all three cases above, two levels of verbosity can also be turned on. For the shell variable cases (1 & 3), append a 'v' or 'V' for minimum or maximum verbosity respectively. For the dhcp command, -v or –V has similar meaning.

6.4.1 Filename Specification in the Server

First of all, note that the BOOTFILE entry in the server's dhcpsrvr.cfg file could have been omitted and the BOOTP transaction would have ended without a file transfer. Depending on the needs of the system, this may be adequate.

Second, the filename specified in the dhcpsrvr.cfg file was a simple file name (no comma delimiters). MicroMonitor transferred the file and because of the lack of any flags, it automatically assumed that the file was an executable script (note the 'e' flag in the output of "tfs ls"). The server could have been set up to transfer file flags and optionally an info field. This is useful for the case where the file is a binary executable rather than a script (the 'E' flag would be used instead of 'e'). For example, if the file was some binary executable and we wanted to load the info field with the release number, then the BOOTFILE line in the dhcpsrvr.cfg file could have used the filename syntax discussed in section 4.2.5 above…

```
BOOTFILE:         some_app,E,rel04
```

The binary executable "some_app" would be stored in TFS with the flag 'E' and the info field "rel04". The output of "tfs ls" would look like…

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 monrc                       73  0xff88005c  e
 some_app                  1245  0xff8802bc  E      rel04
```

6.5 A DHCP Example

This section is going to be similar to section 6.4 above because generally speaking, DHCP and BOOTP are similar. DHCP came about as a result of the shortcomings of BOOTP[38]; hence it offers some additional features. As we did at the start of section 6.4 above, clean up the target's files and environment. At the uMON> prompt, issue the command "tfs rm FNAME" (where FNAME is a filename) for each file listed by "tfs ls" except for monrc. Next, issue the command "set –c" at the uMON> prompt so that all shell variables are cleared from the current environment. At this point the file system and environment are "clean". Refer to Listing 15 for a dump of the output.

Now we need to also modify the dhcpsrvr.cfg file, so terminate the running server (ctrl-c at the shell under which it is running). We want the server to respond to DHCP requests, so change the string `BOOTP_CLIENT_MAC` to `DHCP_CLIENT_MAC`. Now restart the server by typing dhcpsrvr –T again. Finally, issue the DHCP request at the client by typing the command "dhcp", this time without the –b option. At first glance, the output appears essentially the same as with BOOTP. This makes sense since it's basically doing the same thing as the BOOTP example. Now observe the environment by typing "set" at the uMON> prompt.

```
uMON>set
        BOOTFILE = my_first_app,eB,test
        BOOTSRVR = 192.168.1.100
           IPADD = 192.168.1.102
         NETMASK = 255.255.255.0
          GIPADD = 192.168.1.1
         TFTPGET = 334
         CMDSTAT = PASS
           COUNT = 20
```

Notice that now the NETMASK and GIPADD shell variables are also set. These IP addresses are some of the additional bits of information retrieved by DHCP through DHCP options. This is only a small part of the additional DHCP features, and it is certainly beyond the scope of this text to discuss all of them. There are

[38] For more details on the differences between BOOTP & DHCP, refer to the RFCs.

additional features in DHCP that the MicroMonitor client takes advantage of, and for a more detailed discussion, refer to the DHCP command.

6.6 DHCP Coordinated with a Startup Script

In the two examples above the dhcp command was used to kick off the DHCP/BOOTP transaction. For both of those examples, the DHCP transaction was started with the client issuing the DHCP_DISCOVER. With the dhcp command (assumed to typically be launched from within a startup script) no additional retries are made if a server does not reply. It is left to the script to decide how to deal with retries. This provides a bit of flexibility if needed at the time of the DHCP transaction, because if the dhcp command didn't return until the transaction either completed or timed out, it could potentially take several minutes to complete (refer to DHCP RFC). The DHCP RFC specifies a retry/timeout algorithm that can potentially take several minutes, depending on the response (if any) from the DHCP server, and this may or may not be acceptable for your embedded application.

To support a full DHCP-compliant retry mechanism, an alternate means is provided in MicroMonitor to startup via DHCP. If, in the monrc file, the IPADD shell variable is set to DHCP or BOOTP, then the transaction will automatically be started and the retry protocol specified by the DHCP RFC will be used. In either case (startup via the dhcp command in a script or via the IPADD shell variable setting in monrc), the 4-step handshake is done in the background while MicroMonitor is potentially doing other things. This section will demonstrate how the DHCP transaction can be monitored by a script; thus, allowing the script to be coordinated with the completion of the handshake.

Recall that the DHCP transaction will (among other things), establish the content of the IPADD shell variable. This can be used by the script to determine when it can assume the transaction has completed or timed out. To demonstrate this, install the following script on your target…

```
# Autobootable script that kicks off a DHCP transaction and
# has control over DHCP retries...

# Clear the IPADD shell variable...
set IPADD

# Issue the DHCP DISCOVER...
dhcp -v

# Wait for completion of the transaction...
set WAITCOUNT 5
# DHCP_WAIT:
sleep 1
if $IPADD sne \$IPADD goto DHCP_DONE
set WAITCOUNT=$WAITCOUNT-1
if $WAITCOUNT le 0 goto DHCP_GIVEUP
goto DHCP_WAIT

# DHCP_GIVEUP:
echo Too bad! Apparently no server available.
exit

# DHCP_DONE:
echo Cool!  The transaction completed successfully
echo My IP address is: $IPADD
exit
```

Modify the dhcpsrvr.cfg file by preceding the BOOTFILE entry with a pound sign (i.e. #BOOTFILE), then save that file. Before starting up the dhcp server, run the above script…

```
uMON>dhcp_script
  DHCP startup (0 elapsed secs)
Too bad! Apparently no server available.
uMON>
```

Notice that after a few seconds the transaction is aborted because the IPADD shell variable was never populated with a valid IP address. Now run the script a second time, but this time with the server active. Note when you start up the server, omit the –T option (we are not doing any TFTP file transfer in this case)…

```
uMON>dhcp_script
  DHCP startup (0 elapsed secs)
  DHCP request
  Dhcp/Bootp SetEnv: BOOTSRVR = 192.168.1.100
  Dhcp/Bootp SetEnv: IPADD = 192.168.1.102
  Dhcp/Bootp SetEnv: NETMASK = 255.255.255.0
  Dhcp/Bootp SetEnv: GIPADD = 192.168.1.1
Cool! The transaction completed successfully
My IP address is: 192.168.1.102
uMON>
```

Notice that this time the transaction completed quickly (because the server responded) and the script was able to detect the success of the transaction. This demonstrates the flexibility of the combination of script and DHCP startup within MicroMonitor. If the monrc file had established the IPADD shell variable to contain the string "DHCP", then a similar script could have been used to monitor the RFC-compliant DHCP retry mechanism. In either case, application specific commands could be put in the sections of the script to deal with the situation.

6.7 Network Boot without DHCP or BOOTP

Generally speaking when we think of a network bootup we think of DHCP and BOOTP, and usually that's accurate. However, there are times when the target has its IP address but still needs to be told what to do at startup. Also, depending on the network configuration, it just may not be feasible to have a lot of embedded system targets using DHCP to establish their configuration. In that case, the target is assigned a static IP address in the monrc file and TFTP can be used to transfer scripts to the target for startup. The script can be used in several different ways. For example, it can be transferred to the target so that the target can determine whether or not the currently installed application is up-to-date. If it is, then the application is launched, if it isn't the a second tftp transfer can load a new application…

```
01: # TFTP Startup script example.
02:
03: # Make the initial tftp request...
04: tftp -F tftp_script_dld -fe 192.168.1.100 get tftp_script_dld
05:
06: # If the tftp transaction fails, then run the
07: # local copy of the application, else run the
08: # downloaded script...
09: if $TFTPGET seq \$TFTPGET goto RUN_LOCAL
10:
11: # If we get here, then the TFTP transaction must have completed
12: # successfully, so we can run the startup_test script...
13: tftp_script_dld
14: exit
15:
16: # RUN_LOCAL:
17: my_app
18: exit
```

• Listing 19 : Phase 1 TFTP Startup Script

This first script (Listing 19) is the one assumed to be installed on the target as an autobootable script. When the target is reset, the script starts up a TFTP transfer to attempt to retrieve the script "tftp_script_dld" (line 04). If the script is transferred from the host successfully, then it can run (line 13); otherwise, it must be assumed that there is already a local copy (in TFS) of the application and it will be run (line 17).

```
01: # This script is downloaded to the target to test to
02: # see if the version of the "my_app" application is
03: # up to date.  If it is, then just run it.  If it
04: # isn't then get the latest, and then run it...
05:
06: tfs info my_app FINFO
07: if $FINFO seq rel1_4 goto RUN_LOCAL:
08: tftp -F my_app -fe -i rel1_4 192.168.1.100 get my_app
```

```
09:
10: # RUN_LOCAL:
11: my_app
12: exit
```

• Listing 20: Phase 2 TFTP Startup Script

This second script (Listing 20) is downloaded and run by the first script (Listing 19). This script tests to see if the application (in this case, "my_app") is the latest version. The version check is done by comparing the "info" field of the resident file to the string in the script. Line #06 loads the shell variable FINFO with the information filed of the file "my_app", then the if statement of line #07 tests to see if that info field is "rel1_4". If it is then the local copy of my_app can be run; otherwise, a second tftp transfer is started to download the latest version of my_app[39].

Notice that this allows the target to essentially be unaware of the release update. The host running the TFTP server can update its version of my_app and then update the tftp_script_dld script to look for the new release number. After that, all that has to be done is for the target to be reset. Everything else just works, and the target has an up-to-date application.

One final note on this technique… This strategy satisfies the case where a lot of targets are running in a facility and they must be automatically updated occasionally. The files on the server can be updated, then the targets just need to be reset. Upon completion of the scripts above, the targets are updated and the application isn't even aware of it. Now, suppose you want the ability to have certain targets download certain versions of the application… Line #4 of Listing 19 can be changed to

```
tftp -F tftp_script_dld -f e 192.168.1.100 get tftp_script_dld_$IPADD
```

and now you have a unique startup script per target IP address because the requested filename is built with the target's IP address as part of the name. To illustrate this, just try it…at the uMON> command line type "echo hello" then type "echo hello_$IPADD" notice that the first output is simply "hello" and the second one is "hello_192.168.1.102".

6.8 Wrap Up

This chapter demonstrated a variety of different ways MicroMonitor can be used to automatically configure and startup an application from the network. Standard DHCP, BOOTP & TFTP along with MicroMonitor's scripting facilities provides a lot of flexibility. In some cases it may only be necessary for the target to retrieve its own network identification and BOOTP may be sufficient. In other cases, it may be appropriate to use DHCP and TFTP, and it yet others the target may have its ID, but needs the ability to be automatically updated based on files on a server.

The next chapter will walk through some of MicroMonitor's scripting facilities, then you may want to come back to this chapter to see how you can create your own derivative of the examples just discussed.

[39] One important note regarding this technique… The top level script (Listing 19) calls a second script (Listing 20) which then calls a third script (my_app). This uses some stack space in the monitor and depending on how your running monitor was configured, this may cause the monitor to overflow its own stack. Just be aware of this, and if you decide that you are going to use something like this in a real application, then just build the monitor with a larger stack size.

Chapter 7 Writing MicroMonitor Scripts

Since there is a file system and a command interpreter, a fairly easy next step is to have the ability to create executables as simple ASCII files that invoke the commands that are part of the monitor's built-in command set. The monitor's ability to do this goes one step further by providing the ability to do conditional branching, variables, and subroutines. Note that it is not the intent of this chapter to go into full detail on each of the commands used within a script (refer to Chapter 15 for that detail); rather to provide an overall understanding of how to use/write scripts to be run under the monitor. The best way to learn how to take advantage of this capability is through some examples.

7.1 Script Invocation

A file is recognized by TFS as a script if it has the executable (e) flag set[40]. Scripts can be started up in two different ways. Like a UNIX or DOS shell, the script name can be typed on the command line directly and the command interpreter will find it. This implies that a script should not have the same name as any of the built-in commands in the command table because the command interpreter will first look through its own list of built-ins for a match[41].

Alternatively, the "tfs run" command can be used. Running the script this way eliminates the concern of an executable having the same name as an internal built-in, plus it allows the user to specify that the script run with verbosity[42] enabled. With verbosity enabled, each line is printed prior to its execution. For example, in the following script…

```
echo This is a script.
set COUNT 0
# TOP_OF_LOOP:
set COUNT=$COUNT+1
echo $COUNT
sleep  -m 500
if $COUNT ge 3 exit
goto TOP_OF_LOOP
```

the output without verbosity would be…

```
This is a script.
1
2
3
```

however, if verbosity is enabled (by running the script via "tfs –v run {scriptname}"), the output is as follows…

```
[01]: echo This is a script.
This is a script.
[02]: set COUNT 0
[03]: # TOP_OF_LOOP:
[04]: set COUNT=$COUNT+1
[05]: echo $COUNT
1
[06]: sleep  -m 500
[07]: if $COUNT ge 3 exit
[08]: goto TOP_OF_LOOP
[04]: set COUNT=$COUNT+1
[05]: echo $COUNT
```

[40] Note that this is a lower-case 'e'. The upper-case 'E' also refers to an exectuable; however, the distinction between 'e' and 'E' is that one refers to ASCII script executables ('e') and one refers to binary executables ('E').
[41] As of uMon 1.7, a script can be used to seemlessly replace a built-in command. Refer to section 7.14 for more details on that.
[42] The way verbosity is used when running scripts has changed in uMon1.0, from earlier versions.

```
2
[06]: sleep  -m 500
[07]: if $COUNT ge 3 exit
[08]: goto TOP_OF_LOOP
[04]: set COUNT=$COUNT+1
[05]: echo $COUNT
3
[06]: sleep  -m 500
[07]: if $COUNT ge 3 exit
```

The shell variable SCRIPTVERBOSE can also be used to establish/modify script verbosity. Prior to running each line in the script, the shell variable is tested (see section 14.63) and the verbosity is adjusted accordingly. This allows the script itself to enable/disable verbosity.

7.1.1 Per-Command Error Checking

By default, when running a script, MicroMonitor will check the result of each command it runs, and if unsuccessful, the script will terminate at the point of failure. Each command within the monitor's built-in set can return pass or fail (logged in the CMDSTAT shell variable). The fail state may be due to a syntax error or may be due to something specific to the command (for example, the "mt" (memory test) command will return failure if the memory test fails). If it is desirable for the script to just continue after an error, then the command can be prefixed by a dash ('-'), similar to make. For example, the line:

```
mt -cq -t32 -s1 -vv 0x100000 0x1f00000
```

within a script will terminate the script if an error is detected in the specified memory range. The line:

```
-mt -cq -t32 -s1 -vv 0x100000 0x1f00000
```

will allow the script to continue.

7.2 Script-Specific Commands

The following commands within the monitor's command line interface are applicable only within the context of a script:

IF, EXIT, GOSUB, GOTO, ITEM, READ, RETURN

All commands can be put in a script, but the above sets of commands are only useful when used in the context of a script.

7.3 Script Nesting

Script nesting (one script calling another script) is supported and is limited only by the amount of stack that has been allocated to the monitor. The script runner within TFS calls sub-scripts by simply furthering its depth into its own stack, so the only limitation to script calling depth is the allocated stack size. Note that MicroMonitor attempts to notify the user if it determines that it's own stack has overflowed; however, this detection mechanism is obviously limited because it depends on how the overflowed stack affects the running system.

The environment within a script (i.e. the shell variables) is not part of the script's stack. All shell variables are global. If script A sets VARA, then calls script B which modifies VARA, script A will see that modification when script B returns control to script A.

7.4 Scripts Calling Binary Applications

In many cases, an automatically booted (autoboot) script is used in a uMon based system to start up some binary executable application (could be elf, coff, raw binary, etc…). This implies that the script will do some basic setup of shell variables, etc.., then pass control on to the application. Usually, the transfer of control from the script to the application is the last executed line of the script, and there is no reason for the application to return to the script that called it. This works just fine, and is used quite often.

There are cases where it may be useful to run some small binary executable as a preface to the larger application; hence, the need for the binary executable image to return to the calling script. If a script calls a binary executable application of any kind, things can get a bit more complicated, and what happens depends on the way the application was coded. The only time this really becomes an issue is when the script expects to regain control of execution after the application terminates. If the executable simply returns through the same point from which it was called (the entrypoint of that load image), then it will return to the script runner when complete and the calling script will regain control. If the executable terminates using mon_appexit(), then the inoking script context is lost; hence, cannot continue from the point at which it called the application. This is because when mon_appexit is used, the monitor is being reentered; hence, no stack frame to return through, so the calling context is lost and no further "un-nesting" will be done. Refer to the example application (section 8.4) startup code for a simple example that can be configured to use mon_appexit() or simply return to the monitor.

7.5 Example #1: cleanup

This script can be used to apply some conditions to the "tfs clean" command that is used to defragment the FLASH used by TFS. The script can be run at the command line with or without an argument passed to it, or (if the TFS attribute 'b' or 'B' is set) it will automatically run at system startup.

```
# Cleanup script:
# Used to call tfs clean if TFS is running out of flash space.
# If argument is present, then use it as the threshold; else
# use default of 1000000.
#
set SIZE 1000000                  # Establish default size.
if $ARGC eq 1 goto CLEANUP        # If no args, jump to cleanup; else
set SIZE $ARG1                    # set SIZE to cmd line arg value.

# CLEANUP:
tfs -v freemem freemem            # Determine amount of free memory left
                                  # in TFS and place that value in $freemem.
set -i freemem 0                  # Increment by 0 to change to decimal.
if $freemem gt $SIZE goto DONE    # If $freemem is less than size, run
tfs clean                         # tfs clean to defragment.

# DONE:
```

Notice the use of '#' as an indication that the remaining text on the line is a comment. Also, note the use of "goto tags" such as 'CLEANUP' and 'DONE'. When script execution is in progress, if a 'goto' statement is reached (either standalone or part of an 'if' statement) the argument to the 'goto' is assumed to be a tag that exists somewhere else in the script. A target of a goto tag is seen by the monitor as the first white space delimited block of text after a pound sign. The content of the argument to goto is compared to the same number of characters at the start of the tag. As a result, the line

```
# CLEANUP:
```

is the target of the line

```
if $ARGC eq 1 goto CLEANUP
```

even though the target has the colon at the end, the first 7 characters match.

Finally, note that multiple targets with the same text can cause undefined results. The following two lines are considered to be the same target...

```
# CLEANUP: this is the start of tfs defragmentation
# CLEANUP is the first line of tfs defragmentation
```

7.6 Example #2: ping

The following example is similar in syntax to the above script, but carries out a totally different task...

```
#
```

```
# ping script using icmp echo:
# script syntax: ping IP_ADDRESS [optional ping count]
#
if $ARGC eq 2 goto PING_1
if $ARGC eq 3 goto PING_N
echo $ARG0: requires IP address
exit

# PING_1:
icmp echo $ARG1
exit

# PING_N:
icmp -c $ARG2 echo $ARG1
```

Not much to say here, hopefully this is somewhat intuitive. Same kind of deal as cleanup above. Uses comments and goto tags to build a fairly self-explanatory wrapper around the 'icmp' command in the monitor.

7.7 Example #3: namelist

The following example demonstrates a usage of the nested shell variable capability in the monitor's command interpreter...

```
# Build a name list:
set NAME_1 Jane
set NAME_2 John
set NAME_3 Peter
set NAME_4 Paul
set NAME_5 Tommy
set NAME_6 Adam
set NAME_7 Eric
set idx 1
set max 7

# Now print the name list using the idx shell variable as an index:
# TOP:
if $idx gt $max goto DONE
echo ${NAME_${idx}}
set -i idx
goto TOP
# DONE
```

The output of this script would be...

```
Jane
John
Peter
Paul
Tommy
Adam
Eric
```

Note that the 'idx' shell variable is used like an index into an array of names; where the array is called 'NAME_".

7.8 Example # 4: namelist using "item" command

The following example demonstrates a usage of the "item" command in the monitor's command interpreter. The end result is similar to the above example, but implemented in a simpler way...

```
# Print each name in a list:
set idx 1
# Top_of_Loop:
```

```
item $idx NAME Jane John Peter Paul Tommy Adam Eric
if $NAME seq \$NAME exit
echo $NAME
set -i idx
goto Top_of_Loop
```

The output of this script would be...

```
Jane
John
Peter
Paul
Tommy
Adam
Eric
```

7.9 Example # 5: processing a variable number of command line arguments

This example uses the command interpreter's shell variable processing to handle a command line that has a variable number of arguments...

```
# This script processes a variable number of command line
# arguments using nested shell variables...
set idx 0
# TOP:
echo Arg $idx: ${ARG${idx}}
set -i idx
if $idx lt $ARGC goto TOP
```

If the script was invoked with "script aa bb cc dd", the output would be...

```
Arg 0: script
Arg 1: aa
Arg 2: bb
Arg 3: cc
Arg 4: dd
```

7.10 Example # 6: why would you ever want to do this??...

This example takes the shell variable usage to a bit of an extreme, but demonstrates its capability within a script yet again...

```
set abcX HELLO
set defY MOM
set var1 X
set var2 Y
set HELLO_MOM BINGO!
echo ${abc${var1}}_${def${var2}}
echo ${${abc${var1}}_${def${var2}}}
```

The output would be generated by the two final "echo" lines...

```
HELLO_MOM
BINGO!
```

7.11 Example # 7: startup script using subroutines, if/else and file decompression

This is a practical script example that could be used for a system that may (or may not) have the application compressed and their monitor has been configured in such a way that file decompression requires temporary monitor heap expansion. Note that the use of the 0x44000002 is PowerPC specific, but could be re-worked for other CPUs. See the heap man page (section 15.18) for discussion on heap expansion.

```
#######################################################
```

```
#
# run:
# Autobootable script used for starting up MicroMonitor
# based systems.  This script allows the user to be
# unaware of the fact that the application program may be
# compressed. The heap expansion step is needed for decompression
# here because the monitor was intentionally built with less
# heap space allocated to it than is needed for decompression.
# If the monitor was allocated more memory at build time, this
# step could be eliminated.
# In addition, if the BREAKPOINT shell variable is set, then
# after the application has been loaded, but before execution
# starts, the value of 0x44000002 is placed in the address
# specified by $BREAKPOINT.  This inserts an SC (system-call)
# instruction at that address and will cause an exception to
# occur.  At that point, the monitor command "strace" can be
# used to dump the stack frame.
#
# Main:
if $ARGC eq 1 gosub APPNAME_DEFAULT else gosub APPNAME_ARG1
gosub FILE_CHECK
if -t iscmp $APP gosub EXPANDLOAD else gosub NORMALLOAD
if $BREAKPOINT sne \$BREAKPOINT gosub SETBREAKPOINT
call $ENTRYPOINT
reset

# end Main
#
############################################################
#
# begin Subroutines:

# SETBREAKPOINT:
pm -4 $BREAKPOINT 0x44000002
return

# EXPANDLOAD:
heap -X 0xf0200000,0x40000
gosub NORMALLOAD:
heap -x
          return

# NORMALLOAD:
tfs ld $APP
return

# APPNAME_ARG1
set APP $ARG1
return

# APPNAME_DEFAULT:
if $PLATFORM seq MY_PLATFORM goto APP_MINE
if $PLATFORM seq YOUR_PLATFORM goto APP_YOURS
echo Invalid platform: $PLATFORM
exit

# APP_YOURS:
set APP you
return

# APP_MINE:
set APP me
return
```

```
# FILE_CHECK:
set SIZE
tfs size $APP SIZE
if $SIZE sne \$SIZE return
echo File error: $APP
exit

# USAGE:
echo Usage: $ARG0 [appname]
exit
```

7.12 Example #8: Retrieving and Displaying a Bitfield Within a Memory Location

This example demonstrates the use of the –v option in the "dm" command and the expression evaluation built in to the "set" command. Refer to the corresponding manpages for more detail. The script retrieves and processes a register in memory. The variable ESR is loaded with the data stored in location 0x40030009, then a 3-bit field within $ESR is extracted and copied to the variable BCS. Next, the BCS variable is compared to one of 7 different possible values and the resulting branch prints out a verbose description of the bitfield…

```
dm -v ESR 0x40030009
set BCS=hex($ESR&0x70)
if $BCS eq 0x00 goto NOT_USED
if $BCS eq 0x10 goto SLAVE_SELECTED
if $BCS eq 0x20 goto SLAVE_TRANSFER
if $BCS eq 0x30 goto MASTER_TRANSFER
if $BCS eq 0x40 goto FREE_I2C_BUS
if $BCS eq 0x50 goto BUSY_I2C_BUS
if $BCS eq 0x60 goto UNKNOWN_I2C_BUS
if $BCS eq 0x70 goto WAIT_STATE
exit

# NOT_USED:
echo NOT USED
exit

# SLAVE_SELECTED:
echo SLAVE_SELECTED
exit

# SLAVE_TRANSFER:
echo SLAVE_TRANSFER
exit

# MASTER_TRANSFER:
echo MASTER_TRANSFER
exit

# FREE_I2C_BUS:
echo FREE_I2C_BUS
exit

# BUSY_I2C_BUS:
echo BUSY_I2C_BUS
exit

# UNKNOWN_I2C_BUS:
echo UNKNOWN_I2C_BUS
exit

# WAIT_STATE:
echo WAIT_STATE
exit
uMON>
```

7.13 Overriding the Default Command Interpreter

By default, scripts are run in the monitor using the command interpreter that is part of the monitor. This allows all of the commands within the command table of the monitor to be accessible by a script. If an application takes over the system, it may have its own command interpreter, so if the application expects to be able to use the monitor's script runner, then two things must be done:

The application must inform the monitor that it has a command interpreter and that the script runner in the monitor should use that command interpreter instead of the one in the monitor. This allows the monitor's script runner to access the commands that are in the application's command table. This is done with the monitor's API function tfsctrl() and the TFS_DOCOMMAND request (see section 16.49).

The application's command interpreter must provide a hook so that if the command is not seen in the application's command table, the command is passed to the monitor's command interpreter. This can be done within the application's command interpreter by calling the monitor's API function docommand().

If these two steps are taken, then an application will have the ability to use all of its commands (plus the commands in the monitor) in a script that is run when the application is active…

```
01: #include "string.h"
02: #include "monlib.h"
03:
04:
05: int
06: appdocommand(char *line, int verbose)
07: {
08:     if (strcmp(line,"myexit") == 0) {
09:         mon_printf("Gotta go!\n");
10:         mon_appexit(0);
11:     }
12:     if (strcmp(line,"abc") == 0) {
13:         mon_printf("This is my 'ABC' command\n");
14:     }
15:     else if (strcmp(line,"help") == 0) {
16:         mon_printf("exit: terminate the application\n");
17:         mon_printf("abc:  print 'ABC'\n");
18:     }
19:     else {
20:         return(mon_docommand(line,verbose));
21:     }
22:     return(CMD_SUCCESS);
23: }
24:
25: int
26: main(int argc,char *argv[])
27: {
28:     char line[80];
29:
30:     mon_tfsctrl(TFS_DOCOMMAND,(long)appdocommand,0);
31:     while(1) {
32:         mon_printf("MYCLI:");
33:         if (mon_getline(line,sizeof(line),1) > 0)
34:             appdocommand(line,0);
35:     }
36:     mon_printf("Returning control to MicroMonitor...\n");
37:     return(0);
38: }
```

• Listing 21 : Override Default Command Interpreter

First of all, the source code of Listing 21 is getting a bit ahead of us. We haven't built an application yet, nor have we talked about any of the monitor API, so if you're walking through this tutorial, realize that this is just a bit out of sequence. I put this here because of it being related to script running, so keep that in mind, and know that later chapters discuss building applications on top of the MicroMonitor platform in good detail. That being said, the code gives a simple, but fully functional example of overriding the monitor's default command interpreter so that the application's command interpreter can use the monitor's scripting capabilities. First note that the function appdocommand() (lines 5-23) is our "application-specific" command interpreter. Notice that it parses the incoming string checking for command matches and if there is no match within the local commands, the string is passed to the monitor's command interpreter using mon_docommand(). Also, notice the calling parameters of appdocommand(char *line, int verbose). For this strategy to work, the one requirement is that the application's command interpreter function have the same calling parameters as the monitor's command interpreter function.

Now, in main(), prior to entering the loop that waits for commands, the mon_tfsctrl() function (line 30) is used to tell the monitor that the script runner code in the monitor should use appdocommand() instead of the default command interpreter used by the monitor itself. After that call has completed, the application enters a simple command processing loop (lines 31-35). All of the application specific commands (myexit, help & abc) are accessible, plus, thanks to the mon_docommand() call at the bottom of the appdocommand() function (line 20), all of the monitor commands are also accessible. The added capability provided by the call to mon_tfsctrl() is that the application can also call a script and the script execution will use appdocommand() as the command interpreter's entrypoint. This allows the "abc", "help" and "myexit" commands to be included in the list of commands that the script runner knows about. Also, notice that there is a "help" command in the application's set of commands. There is also a "help" command in the monitor's command table. The monitor resolves this in it's command interpreter by looking for a special case, leading underscore. If the first character of the command line is an underscore, '_', then it is dropped and the command line processing starts at the next character; hence, "help" in the application's command set is accessible as "help" and "help" in the monitor's command set is accessed as "_help". After reading through Chapter 8, you may want to come back to this and try it out.

7.14 Replacing a Built-In Command with a Script

As of uMon1.7, it is possible to essentially replace a built-in command with a script. This may be a convenient thing to do for cases where you have a monitor installation and find that one of the built-in commands needs a new feature or has a bug that needs to be fixed; however, it is unrealistic to consider a complete monitor upgrade.

For the sake of this discussion, we'll use an unrealistic example just to demonstrate the capability of the command interpreter. Let's assume that the "dm" command needs to ignore all arguments and simply print out the string "Hi, my name is Bill" instead of its current functionality. It's a 2 step process: 1. create a script with the same name that performs the desired funcionality; and 2. disable the built-in command to be replace. First lets look at the script that will be used to replace "dm"…

```
uMON>tfs cat dm
if $ARG1 seq help goto HELP
echo "Hi, my name is Bill"
exit

# HELP:
echo "Usage: dm"
echo "This command simply prints 'Hi, my name is Bill'"
exit
```

The script covers the case where the first argument is “help” as well as any other cases that may be needed to implement the command (in this case none). Now execute the commands “dm 0” and “help dm” and note the typical output from the built-in dm command…

```
uMON>dm 0
00000000: 0d 00 00 ea e9 00 00 ea   08 01 00 ea c7 00 00 ea   ................
00000010: a6 00 00 ea 69 01 00 ea   28 01 00 ea 47 01 00 ea   ....i...(...G...
00000020: 00 6c 01 00 30 30 3a 32   33 3a 33 31 3a 32 35 3a   .l..00:23:31:25:
00000030: 30 30 3a 31 65 00 ff ff   ff ff ff ff cc 01 9f e5   00:1e...........
00000040: 11 0f 0f ee 00 00 a0 e1   00 00 a0 e1 00 00 a0 e1   ................
00000050: 78 00 a0 e3 10 0f 01 ee   00 00 a0 e1 00 00 a0 e1   x...............
00000060: 00 00 a0 e1 00 00 a0 e3   17 0f 08 ee 17 0f 07 ee   ................
00000070: 9a 0f 07 ee 00 00 a0 e1   00 00 a0 e1 00 00 a0 e1   ................
uMON>
uMON>help dm
Display Memory
Usage: dm -[24bdefl:msv:] {addr} [byte-cnt]
Options:
 -2   short access
 -4   long access
 -b   binary
 -d   decimal
 -e   endian swap
 -f   fifo mode
 -l#  size of line (in bytes)
 -m   use 'more'
 -s   string
 -v {var} load 'var' with element

Required user level: 0
uMON>
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 dm                         148  0x000c067c  e
 monrc                       88  0x00040cdc  e

Total: 2 items listed (236 bytes).
uMON>
```

Although the script exists in TFS and is marked as an executable, it isn’t executed (or seen by help) simply because it has the same name as the built-in and the built-in takes precedence in this case. Now, to essentially eliminate the existence of the built-in ‘dm’ command simply turn it off by specifying a user level of “off” with the ulvl command…

```
uMON>ulvl -c dm,off
```

Now the built-in is essentially disabled and future invocations of ”dm” (for both the command and the help text) will reference the executable script called ”dm”…

```
uMON>help dm
Usage: dm
This command simply prints 'Hi, my name is Bill'
uMON>dm
Hi, my name is Bill
```

```
uMON>
```

With the “dm” built-in command turned off, it can only be restored through a reset of the monitor. It cannot be turned back on any other way. This allows you to make the “switch” in monrc and be guaranteed that it stays switched.

7.15 Wrap Up

MicroMonitor’s script-specific commands are basic; however they provide enough flexibility to do most of the things that are applicable at this level: conditional branching, loops, user and target interaction. Aside from providing alternatives for startup configuration, the simple looping capabilities come in quite handy for hardware debug and diagnostics as well.

Chapter 8 MicroMonitor's Connection to the Application

In much of the discussion regarding the monitor, there is distinction made between the *application* and the *monitor* code. This is done simply to make it clear that the monitor and the application are two totally isolated programs as far as their linkage is concerned. In general, if the monitor is modified, there is no need to rebuild the application; and similarly, if the application is modified, there is no need to rebuild the monitor. They manage to live and cooperate with each other on the same target despite the fact that they are unaware of each other's existence at link time. Compare the monitor and the application program to DOS/BIOS and some application program. DOS & BIOS, without application programs don't have much value, but when you combine them, they compliment each other very nicely. DOS & BIOS provide the "platform" for the application just as the monitor provides a platform for the embedded program.

Ideally, the monitor and the application are thought of as standalone programs, meaning that neither of them need the other to run. This is *almost* true. The monitor is truly standalone. It boots the target system and interfaces to the user through a console and/or network connection. The application, on the other hand, *may* be standalone. It can be installed into the memory of the target system and can be designed to assume nothing regarding the environment provided to it by the monitor. Sometimes this is desirable, and sometimes it is preferred that the application use the facilities of the monitor..

In a typical MicroMonitor based system, the monitor provides a platform that the application can assume exists when it starts up. This platform includes several very useful capabilities that can be taken for granted by the application. For example, in a MicroMonitor based system, the application doesn't have to know anything about field upgradeability, it just comes with the monitor. Compare this to a program on a DOS machine. That program doesn't know how it will be put on the PC, it just knows that it has to run on the PC. The underlying PC, with its disk drive and communication interfaces will take care of getting the program onto the PC. MicroMonitor provides the same thing.

So, the relationship between monitor and application is complimentary. The standalone monitor doesn't do much application-specific stuff; however, when you put an application on top of a monitor-based embedded system, the two live very happily together.

8.1 The Monitor-to-Application Connection

Like an application/BIOS relationship, the application/monitor relationship can vary. The application can use the functionality provided by BIOS, or it can be clever and get around it. Similarly, the application that runs out of TFS through the monitor can use certain monitor facilities, or it can totally ignore the fact that the monitor is present, and execute 100% on its own.

In most cases, the facilities provided by the monitor (particularly TFS) are considered useful in application space, so the application must "connect" to the monitor. For each target, there is some "well known address" provided by the monitor so that the application can make the connection. This well known address is a location that is assumed to contain a pointer to the moncom() function within the monitor (in moncom.c under umon_main/target/common). In addition, the application includes one object file that is built from a file that is considered one of the "common" blocks of code within the monitor, monlib.c. This file could just as easily be compiled by the application build, but it is kept as part of the monitor because it is generic code, independent of the target or application that uses it.

The first call that must be made by the application is a call to monConnect(ulong addr, void (*lock)(), void (*unlock)()). This function (in monlib.c) must be passed the "well-known-address" and optionally, two additional function pointers that will provide the monitor interface with an application-provided mutual-exclusion mechanism (more on that later). For basic connection between monitor and application, the second and third arguments to monConnect() can be NULL. After monConnect() returns, it is then safe for the application to assume that the whole set of "mon_" API functions are available for use. For a detailed discussion on the use of application-provided lock and unlock functions, refer to the section: "Use of Application-Provided Lockout for the Monitor API" (section 8.8).

8.2 Application-Provided Functionality

So far in this chapter, the discussion has been with regard to how the application can hook up to the monitor so that the application can use some of the monitor's functionality. This section will discuss the fact that the monitor can be given some function pointers so that it can use the facilities of the application.

When the monitor is running stand-alone, it has established its own driver interface to the target hardware. For example, it calls its own function to write a character to the console (UART) port. When an RTOS-based application takes over the hardware, it usually re-establishes these interfaces so that the drivers are interrupt driven through the RTOS, not polled through MicroMonitor. Now, while in your application, assume you want to call the mon_docommand() API function to access one of the monitor's CLI commands; however, now the monitor's console I/O functions (i.e. putchar(), getchar(), etc..) are not legal because the application has re-initialized that interface. The solution to this problem is to provide the monitor with a pointer to the application's versions of these functions so that monitor-based console I/O will still interface to the console. Notice in monlib.c that all of the calls to moncom() are "GET_XXX" functions. This is the application "getting" connections to each of the "mon_" functions. The file monlib.h contains all of these #define definitions, but notice that it also has other macros that are not "GET_XXX" type macros. These macros provide the interface to the monitor that allows the application to give the monitor pointers to certain functions...

- CACHEFTYPE_DFLUSH: allows the application to give the monitor a pointer to a data-cache flush routine with the following prototype: dcache_flush(char *addr, int size).
- CACHEFTYPE_IINVALIDATE: allows the application to give the monitor a pointer to an instruction cache invalidate routine with the following prototype: icache_invalidate(char *addr, int size).
- CHARFUNC_PUTCHAR: allows the application to give the monitor a pointer to a putchar function.
- CHARFUNC_GETCHAR: allows the application to give the monitor a pointer to a getchar function.
- CHARFUNC_GOTACHAR: allows the application to give the monitor a pointer to a function that, when called, will return 1 if there is a character queued up on the console port; else 0.
- CHARFUNC_RAWMODEON: allows the application to give the monitor a pointer to a function that will enable "raw" mode on the console port.
- CHARFUNC_RAWMODEOFF: allows the application to give the monitor a pointer to a function that will disable "raw" mode on the console port.

For example, a call to mon_com(CHARFUNC_PUTCHAR,app_putchar,0,0) will tell the monitor to use the function "app_putchar()" instead of its own putchar function for sending characters to the console port.

8.3 Application-Provided Mutual Exclusion

The monitor claims to be independent of the application running on top of it, and potentially there will be a multi-tasking operating system as part of the application. How does the monitor keep its non-reentrant code from being reentered? This is where the second and third arguments to the monConnect() function come in. Like the other "application-provided" functions, the second and third arguments to monConnect() give the monitor a function to call for locking and unlocking the fact that some piece of application code is accessing a monitor facility. Each of the "mon_" functions is wrapped with a call to the lock() and unlock() functions that are provided to the monitor by monConnect() (refer to mon_putchar() as an example). Ideally then, the functions handed to monConnect() for this purpose, should provide priority inversion protection; otherwise, you must be aware of that (as you would any other use of a semaphore).

In addition to this general mechanism for providing reentrancy protection, section 8.8 below discusses the content of monlib.c and the options to further refine the protection based on your system's needs.

8.4 Application-Monitor Hookup: a small, single-threaded example

The following example shows the code that is needed to establish a basic hookup between a small, single-threaded application and the monitor. The code is compiled/linked with start() being the entrypoint and runs off the stack of the monitor. Note the USE_EXIT definition. Refer to the discussion on scripts calling binary executables (section 7.4) for details on this.

```
#include "monlib.h"

int
```

```
main(int argc, char *argv)
{
        mon_printf("Hey, application %s is running!\n",argv[0]);
        mon_printf("Here are the args...\n");
        for (i=1;i<argc;i++)
                mon_printf("arg[%d] = %s\n",argv[i]);

        return(0);
}

void
start()
{
        char    **argv;
        int argc;
        extern  uchar bss_start, bss_end;
        register uchar  *ramstart;

        /* Clear out .bss space:
         * (not really necessary, done already by the TFS loader)
         */
        ramstart = &bss_start;
        while(ramstart < &bss_end)
                *ramstart++ = 0;

                /* Connect the application to the monitor.  This MUST be done
                 * prior to the application making any other attempts to use the
                 * "mon_" functions provided by the monitor.  Note that the value
                 * of the first argument is the content of the MONCOMPTR shell
                 * variable.
                 */

                monConnect((int(*)())(*(unsigned long *)0x20),(void *)0,(void *)0);

                /* Extract argc/argv from structure and call main():
                 */
                mon_getargv(&argc,&argv);

        #ifdef USE_EXIT
                /* Call main, then re-enter the monitor.
                 */
                mon_appexit(main(argc,argv));
        #else
                /* Call main, then return to monitor.
                 */
                return(main(argc,argv));
        #endif
        }
```

8.5 Application-Monitor Hookup: a VxWorks example

The following example shows the code that is needed to establish a connection between a VxWorks application and the monitor. The majority of this code would be inserted into the usrConfig.c file of the BSP. Note that while this discussion refers to VxWorks, the idea is generically applicable to any RTOS that allows the monitor API to be accessed at runtime.

First of all, VxWorks uses the BOOT_LINE_ADRS definition to point to some location in memory that it can use to configure itself. The address is assumed to contain a string that is parsed during the early stages of VxWorks startup. This address should be set to some location in RAM space that is not used by the monitor or the application. Typically, the application is configured to be loaded somewhere above the value of $APPRAMBASE in the monitor. The BOOT_LINE_ADRS address should be set to some block of space between $APPRAMBASE and the actual starting point of the application. For the sake of this example, assume $APPRAMBASE is 0x10000 and the starting point of the VxWorks application is 0x18000. Then build the

VxWorks application with BOOT_LINE_ADRS set to 0x17000 and have the following added to the monrc file (obviously each board's details will be different, this is just an example)...

```
# Basic set of bootup shell variables:
set ETHERADD 00:60:1d:02:0b:fe
set IPADD 192.168.1.102
set HIPADD 192.168.1.100
set GIPADD 192.168.1.1
set NETMASK 255.255.255.0

# Build the VxWorks BOOT_LINE based on the variables established above...
# The "pm -s" command builds a string starting at the specified address
# The "pm -S" command concatenates a string to the end of the string that
#  starts at the specified address
set BOOT_LINE_ADRS 0x17000
pm -s $BOOT_LINE_ADRS "cpm(0,0)"
pm -S $BOOT_LINE_ADRS " e=" $IPADD
pm -S $BOOT_LINE_ADRS " h=" $HIPADD
pm -S $BOOT_LINE_ADRS " g=" $GIPADD
pm -S $BOOT_LINE_ADRS " tn=target u=anonymous pw=vxworks"
```

The result of these entries in the monrc file is that the location 0x17000 will contain the following string...

cpm(0,0) e=192.168.1.102 h=192.168.1.100 g=192.168.1.1 tn=target u=anonymous pw=vxworks

Notice that the content of the BOOT_LINE_ADRS string is dependent on the content of the shell variables established above.

In the BSP (usrConfig.c in most BSPs), the following code makes some of the initial connections. Typically this first call to monConnect() is done near the top of usrInit() just after the .bss space is initialized (usually this is a call to bzero() in usrInit()). It's important that monConnect be called AFTER this point because the initialization of the .bss space will undo some of the stuff set up in monConnect()...

```
#include "monlib.h"

/* Connect the application to the monitor.  This must be done
 * prior to the application making any other attempts to use the "mon_"
 * functions provided by the monitor. The value of 0x80000010 below is the
 * "moncom pointer", the content of the MONCOMPTR shell variable.
 */
monConnect((int(*)())(*(unsigned long *)0x80000010),(void *)0,(void *)0);

/* Tell uMON to use a few of this application's functions...
 */
mon_com(CHARFUNC_PUTCHAR,myPutchar,0,0);
mon_com(CHARFUNC_GETCHAR,myGetchar,0,0);
mon_com(CHARFUNC_GOTACHAR,myGotachar,0,0);
mon_com(CHARFUNC_RAWMODEON,myRawon,0,0);
mon_com(CHARFUNC_RAWMODEOFF,myRawoff,0,0);
mon_com(CACHEFTYPE_DFLUSH,myDcacheFlush,0,0);
mon_com(CACHEFTYPE_IINVALIDATE,myIcacheInvalidate,0,0);
```

Following are the functions that are referenced by the above calls...

```
void
myRawoff( void )
{
       ioctl(0,FIOSETOPTIONS,OPT_CRMOD | OPT_TANDEM | OPT_7_BIT);
}

int
myGetchar( void )
```

```
{
        char    onechar;

        read(0,&onechar,1);
        return((int)onechar);
}

int
myPutchar(char onechar)
{
        return(write(1,&onechar,1));
}

int
myGotachar(void)
{
        int avail;

        avail = 0;
        if (ioctl(0,FIONREAD,(int)&avail) != ERROR) {
                if (avail)
                        return(1);
        }
        return(0);
}

int
myDcacheFlush(char *addr, int size)
{
        return(cacheFlush(DATA_CACHE,addr,size));
}

int
myIcacheInvalidate(char *addr, int size)
{
        return(cacheInvalidate(INSTRUCTION_CACHE,addr,size));
}
```

Once the application's ROOT task has started, and prior to starting up any other tasks, monConnect() should be called a second time to establish the mutual exclusion protection mentioned above...

```
/*****************************************************************
 * Call monConnect a second time (it was already called in
 * usrConfig.c).  This time, pass in the lock/unlock arguments.
 * Note that this MUST be done prior to starting up any other tasks
 * that may use the monitor.
 * Note also, that if the semMCreate fails, there is no need to
 * call monConnect again, since it was called in usrConfig.c.
 */
monSemId = semMCreate(SEM_INVERSION_SAFE | SEM_Q_PRIORITY);
if (!monSemId)
        printf("Could not create monitor access semaphore.\n");
else
        monConnect(0,monLock,monUnlock);
```

Following are the functions that are referenced by the above calls...

```
/* monLock() & monUnlock():
 * Pointers to these functions are passed to monConnect() so that the
 * monitor is guaranteed not to have any re-entrancy problems.
 */
SEM_ID  monSemId;

void
```

```
monLock()
{
        semTake(monSemId,WAIT_FOREVER);
}

void
monUnlock()
{
        semGive(monSemId);
}
```

Finally, if the application has a command interpreter and wants to be able to hook its command interpreter to the script runner in TFS, do the following...

```
/* Tell TFS to use this application's command interpreter when
 * executing scripts.
 */
err = mon_tfsctrl(TFS_DOCOMMAND,(long)docommand,0);
if (err != TFS_OKAY)
        printf("TFS docommand reassign failed: ",mon_tfsctrl(TFS_ERRMSG,err,0));
```

8.6 Application Installation on a Monitor Based System

The above discussions assume that the application is resident on the target system. There are several different ways the application can be installed on a MicroMonitor based system. How it is transferred and where in the system it is transferred to depends on the project. The transfer protocol will typically be Xmodem for serial-port only targets and TFTP for targets with on-board Ethernet. For more information on these transfer protocols and their use with a MicroMonitor based target, refer to Chapter 4 above. For this discussion we will assume TFTP.

Typically, the build process (we'll walk through a few examples in Chapter 9 below) generates an executable image file that is formatted according to the elf standard[43]. The file contains all of the information needed for a smart program (i.e. loader) to copy sections of the file from the storage memory (file system) to the runtime memory (RAM) prior to start of execution. In addition, the elf-formatted file may also have symbolic information appended to the end of the file for use by debuggers. So, the elf file itself is not what the target system's microprocessor wants to execute; however, it does contain all the information needed to properly set up the execution environment needed by the microprocessor. The point is that the elf file must be converted to the actual instructions and data used by the microprocessor at some point prior to execution of the program that the elf file represents. There are basically two techniques that can be used in a MicroMonitor based system:

- On the host system convert the file from elf format to a binary image needed by the microprocessor and then transfer that image to the target.
- Place the complete 'elf' file (minus the symbol table information) on the target and allow TFS to do the 'loader' step.

Here are a few different ways it can done with MicroMonitor running on the target…

8.6.1 File Preparation, Determining the Map and Entrypoint

Much of this discussion talks about the conversion of a file from the ELF format to the binary data stream needed by the microprocessor for actual execution on the target system. Depending on the toolset you're using, you may be able to do this several different ways. For the sake of this text, we'll mention two different mechanisms for this. One is using the GNU cross tools, and the other is to use the 'elf' command that comes with the MicroMonitor distribution, built under umon_main/host. For either method, we'll assume the input elf file is called "program.elf" and the output file is called "program.bin"…

GNU method:

```
objcopy --output-target=binary --gap-fill 0xff program.elf program.bin
```

[43] ELF stands for "executable and linking format". Note that this discussion refers to 'elf', but other formats (i.e. coff, aout, etc..) apply as well, they just aren't as common.

Using the 'elf' tool:

```
elf –B program.bin program.elf
```

Both methods work equally well, take your pick. Next, you need to get the load address of the program and the entrypoint. This too can be done with GNU or 'elf'…

GNU method:

```
objdump --headers --file-headers program.elf
```

Using the 'elf' tool:

```
elf –m program.elf
```

My personal preference is the 'elf' tool because the output is a bit more readable, but either method will produce accurate information.

8.6.2 Binary Image Copied to RAM:

In this case the elf file is converted to binary on the host and transferred directly to the target's RAM at the base address of the image (specified in the image's linker map file). Once in RAM, the entrypoint of the image (which is not necessarily the same as the base address of the image) can be jumped into using the 'call' command in MicroMonitor. For example, assume we have an image file (say image.bin) whose memory map specifies that it be loaded into RAM at location 0x200000, and the entrypoint of that image is 0x200040. Assuming the IP address of the target is 1.2.3.4, the following steps would be taken to download and run that image:

TARGET COMMAND: fm –c 0x200000 0x100000 0
Clear the memory space that is to be occupied by the image (the value of 0x100000 depends on the size of the image). This step may not be necessary if the application starts up by clearing its own .bss space.

HOST COMMAND: ttftp 1.2.3.4 put image.bin 0x200000
This transfers the image.bin file on the host to memory at 0x200000 on the target. Recall that MicroMonitor's TFTP server recognizes the '0x' prefix on the destination file and treats that as an address instead of a filename.

TARGET COMMAND: call 0x200040
This transfers execution from the monitor to the downloaded binary image at its entrypoint of 0x200040.

The advantage of this technique is that it is not writing to flash for each transfer to the target; however, this also means that each time the application is to be started, it must be reloaded from host to target.

8.6.3 Binary Image Copied to TFS:

As an alternative to the above approach, the image.bin file could be transferred to TFS and a script could be used to automate the transfer from TFS to RAM. For example, using the same assumptions as the previous example, the following steps would be taken to download and run that image:

HOST COMMAND: ttftp 1.2.3.4 put image.bin
This transfers the image.bin file on the host to the same filename in TFS.

TARGET COMMAND: fm –c 0x200000 0x100000 0
Clear the memory space that is to be occupied by the image (the value of 0x100000 depends on the size of the image).

TARGET COMMAND: tfs cp image.bin 0x200000
This transfers the content of image.bin (now in TFS) from flash to address 0x200000.

TARGET COMMAND: call 0x200040
This transfers execution from the monitor to the downloaded binary image.

Note that in this case, after the file is transferred to TFS, the next three commands could be a script, and the script could be made autobootable, so that this procedure is automatically done when the system resets. Also note that the image need only be downloaded once.

8.6.4 Formatted ELF File Copied to TFS:

The most common way the application is stored on the target is to transfer the elf-formatted file directly to TFS. TFS understands elf, so it can then be used to extract the memory map information from the file and transfer the various sections (.text, .data, .bss) into RAM space. The formatted file also contains information about the entrypoint and TFS extracts that and automatically turns over control to the image at that point. For example, in addition to the assumptions of the previous example, assume the file 'image' (no .bin extension) is the elf formatted file on the host, the following steps would be taken to download and run that image:

HOST COMMAND: strip image

This removes all symbol table information from the elf file.

HOST COMMAND: ttftp 1.2.3.4 put image image,E

This transfers the elf-formatted 'image' file on the host to the same filename in TFS. Notice that the destination has an ',E' appended to it. This tells MicroMonitor's TFTP server to store it in TFS with the 'executable image' flag set (refer to section 5.1 above for discussion on TFS's file attributes).

TARGET COMMAND: image

MicroMonitor's CLI (command line interface) first looks for the command in it's set of built-ins, then, assuming a match is not made, it looks through TFS to see if there is an executable file with the same name. This invokes TFS's loader to read the elf section headers in the 'image' file and transfer each section as specified to the appropriate address in RAM. In addition, the .bss section is automatically cleared by TFS. After this is complete, the entrypoint (another snippet of information contained in the headers of the elf file) is jumped into and the application then takes over the system.

Note that this is a much simpler procedure. It allows the user to take advantage of TFS's ability to load the image from TFS file storage space to RAM runtime space all in one step. Alternatively, the above single-step could be broken down into individual load and run commands if so desired…

TARGET COMMAND: tfs –v ld image

This uses the TFS subcommand 'ld' (load) with the verbose flag set to show the user where it is placing each of the sections within the elf file.

TARGET COMMAND: call $ENTRYPOINT

This uses the ENTRYPOINT shell variable as an argument to the call command. The shell variable ENTRYPOINT is automatically created by the "tfs ld" command for use by a subsequent 'call' command. As a result, these two commands could have been put in a script and executed. The purpose behind wanting to split the load and run into two steps is to allow the user (under certain circumstances) to do something after the load but before execution starts. For example, some targets support insertion of a trap or breakpoint-like instruction into the RAM based instruction stream.

8.6.5 Compressed & Formatted ELF File Copied to TFS:

This section is essentially the same as the previous section except that the elf-formatted file is compressed using the 'elf' tool on the host. Then, when stored in TFS on the target, the 'c' flag is included so that TFS knows that the individual sections within the elf file format have been compressed. For example, using the same assumptions as the previous example, the following steps would be taken to download and run that image:

HOST COMMAND: strip image

Remove all symbol table information.

HOST COMMAND: elf –z6 image

Compress each of the sections of the elf image using the zlib compression algorithm (built into the elf tool). Note that if the image's headers are in little-endian, then "elf –cz6 image" is the command to use. This creates a file called image.ezip.

HOST COMMAND: ttftp 1.2.3.4 put image.ezip image,Ec

This transfers the elf-compressed/formatted 'image.ezip' file on the host to the file 'image' in TFS. Notice that the destination now has an ',Ec' appended to it. This tells MicroMonitor's TFTP server to store it in TFS with the 'executable compressed image' flags set (refer to section 5.1 above for discussion on TFS's file attributes).

TARGET COMMAND: image

MicroMonitor's CLI (command line interface) first looks for the command in it's set of built-ins, then, assuming a match is not made, it looks through TFS to see if there is an executable file with the same name. This invokes TFS's loader to read the elf section headers in the 'image' file and decompress/transfer each section as specified to the appropriate address in RAM. In addition, the .bss section is automatically cleared by TFS. After this is complete, the entrypoint (another snippet of information contained in the headers of the elf file) is jumped into and the application then takes over the system.

This final scenario is the ideal case, but requires that the monitor be built with decompression installed. To determine if your monitor has decompression, run the 'help' command and see if 'unzip' is one of the commands. If yes, then decompression is installed.

8.7 Extending the Monitor's Heap

The monitor has its own memory manager (malloc.c & sbrk.c). By default, the monitor is typically configured to have 8K of memory dedicated to the monitor's malloc (actually this varies from one target to the next). There are cases where this may be insufficient and the monitor configuration can be modified so that a larger amount of memory is statically allocated for this purpose. An alternative to this is to extend the space using the -X option of the heap command (or the mon_heapextend() API). This allows the monitor's default amount of memory to remain relatively small, but does not prohibit the ability to increase the size if necessary.

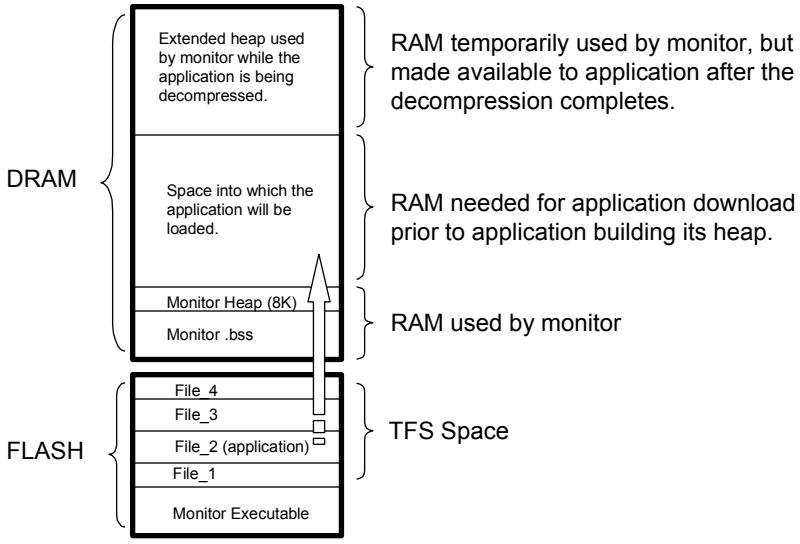
Usually, a project will put some operating system on top of the monitor and use that OS's malloc; however, the monitor code itself still maintains its own small heap independent of the OS. The monitor's heap is used mostly for shell variable storage and a few operations in TFS, which generally don't need much memory. Larger heaps can be made if the application uses the monitor's malloc (via mon_malloc()) and also if the ZLIB code is active for decompression. Following are two examples of situations where heap -X would be used to compensate for the small amount of memory configured for use by the monitor's malloc, but still not require that the monitor have a large heap compiled in. Note that an alternative to the "heap -X" command at the monitor's command line is to use the API call mon_heapextend().

8.7.1 An Application that uses the monitor's malloc:

Usually, an application has its own malloc and when it takes over the target, the monitor's malloc is only used by the monitor. This doesn't have to be the case. A small application that does not need an OS may still need some way to allocate memory temporarily, so mon_malloc() can be used for this. This being the case, then it is likely that the 8K of space compiled in as the monitor's heap will not be sufficient. This is where heap -X {start,size} comes in. In the monrc file, or anywhere prior to the application starting to allocate memory beyond the basic allocation (typically 8K), set up these two parameters. The value of start should be loaded with a starting address from which malloc will get additional memory and size is the size of that block of memory. Once this is done, the monitor's malloc will be able to allocate memory from that block of space if it cannot fulfill a request using its initial block of space.

8.7.2 Using ZLIB to decompress files in TFS:

The use of zlib to decompress is likely to need more than the basic 8K of memory that the monitor's heap is configured with. Like the above example, the heap size can be increased to allow zlib to do its thing. This is good, but once zlib has completed the decompression of the application, it is very likely that heap extension will no longer be needed by the monitor. To compensate for this, the extension can be released from the monitor as long as there is no memory currently allocated in the space that spans the extension. For example, the heap can be extended, the executable decompressed into DRAM, then the heap can be released and control can then be turned over to the application. The only obvious requirement is that the space into which the application is being loaded cannot overlap with the extended heap space. Following is a pictorial view of the extended heap space and a script to extend the heap, load a compressed executable, release the heap and run the executable...


Extended heap used by monitor while the application is being decompressed.
RAM temporarily used by monitor, but made available to application after the decompression completes.
DRAM
Space into which the application will be loaded.
RAM needed for application download prior to application building its heap.
Monitor Heap (8K)
Monitor .bss
RAM used by monitor
File_4
File_3
File_2 (application)
File_1
TFS Space
FLASH
Monitor Executable

Example of Memory Space Allocations for Unzipping an Application to DRAM
(sizes implied by divisions of memory space should be ignored)

• **Figure 4: Monitor Memory Map**

```
# Extend the heap as necessary:
heap -X 0x300000,0x80000

# Load the application, allowing zlib to use the extended
# heap.  (this will load ENTRYPOINT with the entry point of
# the application 'File_2'):
tfs ld File_2

# Eliminate the heap extension:
heap -x

# Transfer control to the entry point of
# the application previously loaded:
call $ENTRYPOINT
```

8.8 Use of Application-Provided Lockout for the Monitor's API

The purpose of this section is to raise the user's awareness of the issues around the use of the monitor's API in a multi-tasking environment. The monitor itself is a single threaded program. There are no interrupts hence, aside from recursion, the code within the monitor does not have to deal with the issue of reentrancy or sharing of resources. However, when the monitor API is used in a multi-tasking environment things get more complicated. Since multiple tasks can access the monitor API, without some type of lock out mechanism, it is very likely that two different tasks may try to use a facility in the monitor that is not prepared for multi-task access.

The primary mechanism to deal with this is to wrap each monitor API function with an application-provided lock/unlock function that will treat the monitor API as a resource and provide mutual exclusion for that resource. For example, we never want to be inside the tfsadd() function more than once. It is very important that this function be protected from reentrancy because it accesses the underlying flash and while the flash is being modified, we don't want to be interrupted by another function that will also attempt to modify the flash. The function in application space that hooks to the monitor's tfsadd() function is called mon_tfsadd(). The code for mon_tfsadd() follows...

```
int
mon_tfsadd(char *name, char *info, char *flags, unsigned char *src, int size)
{
     int ret;
     TFS_MONLOCK();
     ret = _tfsadd(name, info, flags, src, size);
     TFS_MONUNLOCK();
     return(ret);
}
```

All of the monitor API functions look very similar to this one. The _tfsadd() function pointer and the code within monLock() and monUnlock() are all established by the call to monConnect() when the application first hooks itself up to the monitor. Assuming the application has established some lockout mechanism, this guarantees that while tfsadd() is being executed, no other monitor API function will run. As a general solution, this works ok. It is left up to the application to pick the best solution for the lockout (semaphore, mutex, scheduler disable, critical section, etc.). The underlying hardware platform, the monitor API facilities being used and the way they are used by the application are the major factors used to determine which mechanism fits best. The right choice will vary, and it is possible that there will be no need for a lockout mechanism at all. While this solution is generally applicable, it does have some limitations and raise some questions...

All of the monitor facilities use one lockout mechanism, despite the fact that they do not all have reentrancy conflicts.
For example, the monitor facilities used by getenv() do not interfere with those of tfsadd() so it seems like an application should be able to call mon_getenv() even if mon_tfsadd() is in progress.

If one task calls a monitor API function that blocks, then no other task can access a monitor facility until that first API call completes.
If I have one task in my application that calls mon_getline(), then no other task will be able to access a monitor API function until mon_getline() returns. Since mon_getline() blocks waiting for a full line of console input, this will be very inefficient.

If I am inside a monitor API function, I can't call another monitor API function because the first call will lock out the second one.
Assume an application uses mon_docommand() in it's CLI, plus some application-specific commands have been previously installed into the monitor's command table (using mon_addcommand()), and within the code of one of the application-specific commands there is a call mon_printf(). This will lock up because the call to mon_docommand() earlier in the function nest has locked out any further API calls until the return from mon_docommand().

These are all valid concerns, and depending on the situation, they may or may not be an issue for your project. The remaining discussion will attempt to solve the above problems, or at least make it clear to the reader just what the problem is.

First of all, the major "single user" resource in the monitor is the flash device. At first glance, this accurately implies that protection is needed in a multi-tasking environment so that two tasks never attempt to call tfsadd(). This is certainly a valid concern; however, a more subtle issue is the fact that the monitor itself may be executing directly out of the same flash that is used by TFS. Now we have to deal with the fact that while tfsadd() is modifying flash, we do not want ANY OTHER monitor API function to run because it would fetch instructions from the same flash device that has an operation (erase or write) in progress. In generic terms, this is illegal for flash devices. Typically, if a flash operation is in progress, fetching from that same device will fail. This is only a problem if the monitor is fetching instructions from the same device that TFS uses for file storage. There are a few solutions to this, some of which simply require coding discipline and others that require a hardware modification. Certain solutions can make the lock/unlock restriction much more relaxed...

Limit the monitor API function accesses to one task. This entirely eliminates the need for the lockout because as far as the monitor is concerned, it is only being accessed by a single thread.

Do not use any of the blocking monitor API functions (mon_getchar(), mon_getline() and mon_getbytes()). This eliminates the chance that some other task will be indefinitely locked out because one of these functions is blocked and holding the lock.

Configure the monitor so that it runs out of RAM. If the monitor is configured to copy itself into RAM at startup and fetch instructions only from RAM, then operations on flash will not conflict with execution of the monitor in RAM space. In this situation, most of the non-flash related monitor API functions can run without the lock.

Separate the boot flash device from the TFS storage device. The boot flash device may not be the same device that is to be used for TFS file storage. In this case, even if the monitor is fetching from the boot flash, the flash operations (erase and/or write) are being done on some other device; hence no conflict.

Use a flash device that supports simultaneous execution and operation. Typically, these devices are architected such that the device is essentially in two halves. One half can be executed from while the other half is being operated on. Then the only issue is to make sure that the monitor executable is in one half and the TFS storage space is in the other half.

Use a lockout mechanism that disables context switches. If this is done, then only the monitor API functions that deal with flash access need to be wrapped with the lock functions. This is because once the lockout is enabled, no other monitor API function will be callable until that one completes. An extension to this would be to make the lockout function aware of itself, so that it deals with nesting. Following is a generic example of a lockout function that would accomplish this (code contributed by Jim Apgar)...

```
static OSPRIORITY save_pri;
static U32 umon_nest_level = 0;
static void
umon_lock (void)
{
    ENTER_CRITICAL_SECTION();
    if (umon_nest_level == 0) {
        // Raise priority to prevent other processes
        // from interrupting monitor operations
        save_pri = GET_CURRENT_TASK_PRIORITY();
        SET_CURRENT_TASK_PRIORITY(HIGHEST_PRIORITY);
    }
    umon_nest_level++;
    EXIT_CRITICAL_SECTION();
}

static void
umon_unlock(void)
{
    ENTER_CRITICAL_SECTION();
    if (umon_nest_level > 0) {
        umon_nest_level--;
    }
    if (umon_nest_level == 0) {
        // Restore original priority
        SET_CURRENT_TASK_PRIORITY( save_pri );
    }
    EXIT_CRITICAL_SECTION();
}
```

There are probably other solutions, but as you can see from the list above, there are at least several choices. Each one has different pros and cons, so its hard to say which is best. The most important thing is to be aware of your situation so you can make the right choice. As mentioned at the top of this section, the default means of lockout is to assign a single application-specific facility to all monitor API functions. Depending on your solution, you may be able to eliminate the need for all lockouts that are not related to flash. Or, you may find that you need to establish more than one lock out mechanism... one for flash access, one for environment access, etc. In monlib.c, there is a set of macros at the top of the file that establish the default monLock/monUnlock facility. Following is the code and comments taken directly from that source file...

```
/*****************************************************************
```

```
 *
 * The following macros support the default monitor lock/unlock mechanism when
 * they point to monLock and monUnlock.  If something other than the default
 * is to be used, then simply redefine them here.  Refer to the monitor
 * app note that discusses multi-tasking access to the monitor API for more
 * information.
 *
 * TFS_MONLOCK/UNLOCK:
 * Lock/unlock for functions that access TFS flash space:
 */
#define TFS_MONLOCK   monLock
#define TFS_MONUNLOCK  monUnlock

/* ENV_MONLOCK/UNLOCK:
 * Lock/unlock for functions that access monitor shell variables:
 */
#define ENV_MONLOCK   monLock
#define ENV_MONUNLOCK  monUnlock

/* CONSOLE_MONLOCK/UNLOCK:
 * Lock/unlock for functions in the monitor that deal with console output.
 */
#define CONSOLE_MONLOCK  monLock
#define CONSOLE_MONUNLOCK monUnlock

/* HEAP_MONLOCK/UNLOCK:
 * Lock/unlock for functions in the monitor that deal with the heap.
 */
#define HEAP_MONLOCK  monLock
#define HEAP_MONUNLOCK  monUnlock

/* BLOCKING_MONLOCK/UNLOCK:
 * Lock/unlock for functions in the monitor that block waiting for
 * console input.
 */
#define BLOCKING_MONLOCK monLock
#define BLOCKING_MONUNLOCK monUnlock

/* GENERIC_MONLOCK/UNLOCK:
 * Lock/unlock for all functions not covered by the above macros.
 */
#define GENERIC_MONLOCK  monLock
#define GENERIC_MONUNLOCK monUnlock
```

Since monlib.c is actually part of the application, it is 100% legal to make application specific adjustments to this file. These macros should make those adjustments a bit less painful.

Acknowledgements:

Most of this section is a result of discussion with Jim Apgar, a monitor user, who ran into some issues that caused him to deviate from the standard use of the monitor API lockout facilities. Thanks Jim!

8.9 Wrap Up

uMon supports several different modes of binary files; hopefully this chapter gives you a good overview of how to take advantage of each of the modes. You can run a raw binary image, or a formatted (elf/coff etc..) binary image. Your application can be compressed or left uncompressed. Your application can even hook to the monitor and use some of its features when it is running. Lots of options, lots of decisions, all good.

Chapter 9 Binary Application Examples

We've discussed about as much as we can discuss without getting into some code. Now it's time to build a very basic "hello world" application and add the necessary hooks to allow it to take advantage of the fact that it will reside as an application file in MicroMonitor's TFS flash space. The "download and run" portion in this chapter is basically going to be a repeat of stuff we've already covered in earlier chapters. The only difference is that this time the application will be a binary executable ELF file, not just a simple ASCII text script. Note that this will be a basic example. Then, we'll follow this basic example with a few more examples that demonstrate some of the capabilities that an application "inherits" as a result of the underlying MicroMonitor platform. An in-depth discussion of the various ways in which an application runs on top of MicroMonitor was discussed in Chapter 8 above.

This chapter assumes the toolset is version 3 of the Microcross GNU X-Tools package. A beta version of these tools is available on the CD of my book "Embedded Systems Firmware Demystified"[44], or an up-to-date version can be purchased from Microcross. The tools are standard GNU-GCC, so regardless of the tool vendor, the following text is fairly applicable. All MicroMonitor builds are based on the Microcross X-Tools, so using them guarantees a clean build of the monitor and all of the applications discussed throughout this text. In addition to the GNU-GCC tools, there are a few tools that are packaged with MicroMonitor. Some of these can be replaced with equivalent GNU-tools, and some cannot[45].

Each of these "Application #N" sections will build on each other (as N increases), so it is best to read this entire chapter for maximum clarity. Using these examples along with the umon_apps/demo example that comes with the monitor source tree, provides a good working understanding of the structure of an application running on top of a MicroMonitor based target.

9.1 Architecture Independent Configuration

Before diving into the applications, it seems reasonable to start with a bit of an overview of the makefile and the files surrounding each of the applications to be discussed below. The source code in this chapter is found under the umon_apps/user_manual directory (part of the uMon1.0 source code distribution).

Since this documentation attempts to be CPU/target system independent, we need to work in an environment that will deal with that; hence, we can't just assume the target is a PowerPC based CSB472 for example. As of uMon 1.0, a new, more generic set of demo applications comes with the distributed source tree. This allows the user to use a single source directory to build an application that will ultimately run on any target running ARM, ColdFire, PowerPC or MIPS[46]. This is done through the use of a few "make" command line definitions, referred to in the makefile as "site dependent data". Following is a snippet of text from that makefile:

```
# Site dependent information:
# Adjust these values based on your system configuration.
# ARCH:
#      Set ARCH to one of the accepted CPU architectures (i.e. MIPS
#      PPC, ARM, COLDFIRE).
# MONCOMPTR:
#      Retrieve MONCOMPTR from the output of 'help -i' or the content of
#      MONCOMPTR shell variable.
# APPRAMBASE:
#      Set APPRAMBASE to the content of the APPRAMBASE shell variable
#      or a bit higher.
# TARGET_IP:
#      Set TARGET_IP to the IP address of your target.
```

The values supplied for ARCH and TARGET_IP will depend on your target configuration. For the sake of these examples, we will assume the ARCH is MIPS and the TARGET_IP is 192.168.1.110. The values supplied for MONCOMPTR and APPRAMBASE are retrieved from the target by referring to the output of the

[44] The beta version of these tools may not support all of the constructs discussed in this text.
[45] Complete documentation on the tools supplied with MicroMonitor can be found in Chapter 17.
[46] Other CPU support can be added as needed.

"set" command. This command simply dumps the currently established shell variables in uMon, two of which are MONCOMPTR and APPRAMBASE. Using these variables, the makefile automatically builds in the appropriate CPU-specific startup code (which, in some cases, includes the establishment of an application-owned stack) mapped to run in the RAM space of your target (using APPRAMBASE as the memory map base and MONCOMPTR as the hook into your version of uMon).

Prior to working through the following examples, issue the "set" command on your target to retrieve the MONCOMPTR and APPRAMBASE values. Then, assuming you're under the umon_apps/user_manual directory, fill in the entries at the top of the makefile for MONCOMPTR, APPRAMBASE, ARCH and TARGET_IP. These entries will be used for all of the applications discussed below.

9.2 App #1: Embedded "hello world"

9.2.1 main1.c:

This is the MicroMonitor application equivalent of "hello world", consisting of two functions: start and main. These two chunks of code are separated into two functions for clarity. If you really wanted to, you could just do it with one function. The start() function is the entrypoint into the application[47]. It is within this function that the application prepares itself for main(), similar to startup code in crt0.s in some systems. In its simplest form, it attaches to the monitor, retrieves command line arguments and calls main in the familiar *main(int argc, char *argv[])* style.

```
#include "monlib.h"

int
main(int argc, char *argv[])
{
        mon_printf("Hello embedded world!\n");
        return(0);
}

int
start(void)
{
        int    argc;
        char   **argv;

        monConnect((int(*)())(*(unsigned long *)MONCOMPTR),(void *)0,(void *)0);

        /* Extract argc/argv from structure and call main(): */
        mon_getargv(&argc, &argv);

        /* Call main, then return to monitor. */
        return(main(argc, argv));
}
```

• Listing 22 : Application #1 main.c

This is a generic start() function for simple MicroMonitor based applications. It assumes the application is running off of the stack that the monitor has, and does not deal with any C++ constructor/destructor issues. The call to monConnect() (and the inclusion of monlib.c in the makefile) is what establishes the linkage between this application and the underlying monitor.

Referring to Listing 22, just below the call to monConnect(), is the first "mon_" function. All MicroMonitor API functions are prefixed with "mon_" just to make sure their names are kept unique from other standard library functions that may be available. For example, there will be cases where both mon_printf() and printf() are used in an application, so the use of "mon_" not only identifies the function as being part of the MicroMonitor API; but it also provides necessary name isolation from standard libraries. The mon_getargv() function simply retrieves the argc/argv information (if any) from the monitor so that it can be passed to main(int

[47] In most toolsets "start" is the default name of the application entrypoint. For GNU tools, if there is a need to override this, the "-e" option is used with ld.

argc, char *argv[]). The inclusion of monlib.h is necessary to resolve the monitor API calls, and this file requires the presence of two other MicroMonitor source files, cli.h and tfs.h. Finally, the main() function is almost identical to standard "hello world" main, except that "mon_" is a prefix to printf().

9.2.2 Download and Run It…

From the umon_apps/user_manual directory, run "make app1" to build the application. The result is an ELF file called "app1". This file can be directly copied to the target's TFS space with the command[48]:

```
ttftp 192.168.1.110 put app1 app1,E
```

Use the IP address that your target has been configured for (refer to section 2.5 above, if you haven't configured your target's IP address). The ttftp command is used to download the file "app1" to the target. The same destination name is used, but we add the TFS flag 'E' which lets the target know that the image is a binary executable (not a script, which would use the flag 'e').

Now all that's left to do is run it. The name of the application can simply be typed on the command line and TFS will automatically load it and jump to the entrypoint[49]. Assuming all previous steps were followed correctly, the output should be as follows…

```
uMON>app1
Hello embedded world!
uMON>
```

If you want to see what actually happened behind the scenes, you can break the above step up into a few smaller parts. First of all, the "app" file contains the text and data space that is to be copied to DRAM. It also contains enough information to allow TFS to extract the location of the BSS section so that TFS can clear that space prior to jumping to the entry point. Once the text and data are copied and the bss is cleared, then TFS simply calls the location specified by the ELF file's entrypoint. Here are the steps, as they would be at the uMON> prompt…

```
uMON>tfs -v ld app1
.text     : copy    4584 bytes from 0xff88c090 to 0x00020000
.data     : copy       4 bytes from 0xff88d278 to 0x000211e8
.rodata   : copy      23 bytes from 0xff88d27c to 0x000211ec
.bss      : set      256 bytes  at  0x00021204 to 0x00
.comment  :          108 bytes not processed (tot=108)
.shstrtab :           45 bytes not processed (tot=153)
entrypoint: 0x20018
uMON>call $ENTRYPOINT
Hello embedded world!
Returned: 0 (0x0)
uMON>
```

Note that the exact memory map information may vary, but the point is that two commands are run in place of the earlier "app" command…

tfs –v ld app to transfer the sections in the ELF file from flash space to DRAM and also to clear the BSS sections

call $ENTRYPOINT to transfer control from the monitor's command interpreter to the starting point of the application. The ENTRYPOINT shell variable is automatically created by "tfs ld" for this kind of command sequence.

That's it for building about as simple of an application that can be built. The purpose of this section was to provide a basic example of what is done to build an application to reside on top of the MicroMonitor platform. The following sections give a few more examples of what application code can use in the MicroMonitor environment, plus some additional build-time snippets that come in handy.

[48] The makefile has a "dld" target that can also be used for this if the shell variable TARGET_IP is set to the IP address of your target.

[49] MicroMonitor's command interpreter will first look through its list of built-ins for a match, then search through TFS for a match; so, the only limitation here is that the application name should not be the name of a built-in command.

9.3 App #2: Applications Built Using Portions of MicroMonitor Common

This example will use a similar set of files as the first application; however, organization and complexity of each will be increased to develop a template for building real applications that reside on top of MicroMonitor. Also, the application will now startup with an assembly-code based entrypoint that will establish a stack frame that is independent of MicroMonitor..

9.3.1 More Initialization in start()

This new version of start adds a lot more functionality to the pre-main startup code. The code of application #1 ran using the stack of MicroMonitor. When MicroMonitor turns over control to an application, it simply jumps to the application's entrypoint. If the application's startup code chooses not to establish its own stack frame, then it will nest itself into the stack frame of MicroMonitor. This is ok for simple applications that don't require a lot of stack space, and also don't need the monitor's ability to dump a stack trace; however, for most real applications, it is wise that the entrypoint (start() in most cases) establish a stack frame independent of the monitor's stack frame. Also this new Cstart() (called by the stack-initialization assembly code that contains the start entrypoint) clears it's own BSS space.

```
 1  #include "monlib.h"
 2
 3  extern char   _bss_start, _bss_end;
 4
 5  unsigned long AppStack[1024];
 6  extern int main();
 7
 8  void
 9  Cstart(void)
10  {
11      char      **argv;
12      int              argc, ret;
13      register char *ramstart, *ramend;
14
15      /* Initialize application-owned BSS space.
16       * If this application is launched by TFS, then TFS does
17       * it automatically, however since MicroMonitor provides
18       * other alternatives for launching an application, we
19       * clear bss here anyway (just in case TFS is not launching
20       * the app)...
21       */
22      ramstart = &_bss_start;
23      ramend = &_bss_end;
24      while(ramstart < ramend)
25         *ramstart++ = 0;
26
27
28      /* Connect the application to the monitor.  This must be done
29       * prior to the application making any other attempts to use
30       * the "mon_" functions provided by the monitor.
31       */
32      monConnect((int(*)())(*(unsigned long *)0xff800008),(void *)0,(void *)0);
33
34      /* Extract argc/argv from structure and call main(): */
35      mon_getargv(&argc,&argv);
36
37      /* Call main, then return to monitor. */
38      ret = main(argc,argv);
39
40      /* Since we established a stack frame, we can't just return to
41       * the monitor.  We have to exit...
42       */
43      mon_appexit(ret);
44  }
```

• **Listing 23: Typical Application Startup Function**

Lines 22-25 clear the BSS space for this application. In cases where the application is launched by TFS (which when running with MicroMonitor, this is likely), BSS is automatically cleared; however, during development, there are options that allow you to side-step the TFS load process if appropriate. When doing this, it is important to note that TFS is not clearing BSS; hence, the application must do it. Also note that depending on the compiler, the names __bss_start and __end may be different, so if your linkage step complains about these entries, the easiest thing to do (for the sake of this demo) is to just comment out that loop.

Lines 32-35 are identical to those of app #1 (section 9.2.1 above).

Lines #38&43 are necessary for the cases where the application redefines the stack frame. With the stack pointer changed, the return point of this function was lost; hence, a simple return location is undefined. This function allows the application to gracefully return to the monitor; however, the context that was established when the monitor first called the entrypoint (start) is lost.

9.3.2 More Monitor API Calls in main.c

The source code of main.c is shown in Listing 24.

```
 1  #include "monlib.h"
 2
 3  int
 4  main(int argc,char *argv[])
 5  {
 6          int    i;
 7
 8          /* Use MicroMonitor's printf() for standard out...
 9           */
10          mon_printf("Hello embedded world!\n");
11
12          /* The argument list is available just like a standard
13           * application...
14           */
15          for(i=0;i<argc;i++)
16                  mon_printf("argv[%d] = %s\n",i,argv[i]);
17
18          /* If memory trace is enabled, then this statement will be
19           * logged, and readable by the mtrace command...
20           */
21          mon_memtrace("hey, this is a trace statement");
22
23          /* The application can use previously established shell
24           * variables to configure its runtime...
25           */
26          if (mon_getenv("ABC"))
27                  mon_printf("ABC = %s\n",mon_getenv("ABC"));
28
29          return(0);
30  }
```

• **Listing 24: MicroMonitor API Calls In "main.c"**

Lines 15-16 demonstrate the ability to take advantage of normal argc/argv processing as is made available to any application launched from a UNIX or Windows application. The call to mon_getargv() in start.c is what prepares the application for this.

Line #21 is an API call that demonstrates the monitor's ability to record run-time trace that can be looked at some time in the future. The "mtrace" command in the monitor enables this tracing, and a more detailed discussion of it will come in section 10.4 below.

Lines 26-27 demonstrate the fact that MicroMonitor allows the application to retrieve shell variables that were set up prior to the application starting up.

9.3.3 Build, Download & Run

Run "make app2" then "make app2dld" to download the app2 file to the target. At the uMON> prompt of your target, just type "app2". The output should be as follows…

```
uMON>app2
Hello embedded world!
argv[0] = app

Application Exit Status: 0 (0x0)
uMON>
```

Now, at the uMON> prompt, type "app2 1 2 3" and notice that the application detects the presence of arguments…

```
uMON>app2 1 2 3
Hello embedded world!
argv[0] = app
argv[1] = 1
argv[2] = 2
argv[3] = 3

Application Exit Status: 0 (0x0)
uMON>
```

Again, at the uMON> prompt, type the command "set ABC YIKES", then type "app2" and notice that the application detects the presence of the shell variable ABC…

```
uMON>app2
Hello embedded world!
argv[0] = app
ABC = YIKES

Application Exit Status: 0 (0x0)
uMON>
```

Finally, at the uMON> prompt, type the commands "set MTRACEBUF=$APPRAMBASE+0x100000" followed by "mtrace cfg $MTRACEBUF 0x1000", then type "app2". The application's output doesn't appear to have anything additional this time. That's because the trace was logged to the space that was configured for mtrace. Type the command "mtrace dump" and you'll see the log as expected…

```
uMON>set MTRACEBUF=$APPRAMBASE+0x100000
uMON>mtrace cfg $MTRACEBUF 0x1000
uMON>app2
Hello embedded world!
argv[0] = app
ABC = YIKES

Application Exit Status: 0 (0x0)
uMON>mtrace dump

<0001> hey, this is a trace statement

uMON>
```

9.4 Establishing a Stack Frame for Various CPU Architectures

Since we eluded to the establishment of a stack frame in the previous section, we'll very briefly show how to set up an application-owned stack for ARM, MIPS, PowerPC and Coldfire. Note that it is beyond the scope of this document to go into a lot of target-specific detail; however, the basic setup of a non-monitor-owned stack is essential. Each of these examples are included in the demo application code under the uMon source code distribution, they are shown here simply for completeness of the discussion. Each of these startup points uses the "start" tag and assumes that the first 'C' function is Cstart(). They simply establish the stack pointer to be the end of the AppStack[] array, then jump to Cstart(). This sort of pre-C code is generally part of a compiler package called crt0.s; however, it is provide here to provide independence from that linkage.

9.4.1 ARM-Based Stack Setup

```
.extern AppStack
.extern Cstart
.global start

.text

start:
    ldr sp, =(AppStack + 4096 - 16)
jump_to_c:
    bl Cstart
```

9.4.2 MIPS-Based Stack Setup

```
#define sp    $29
#define k0    $26

.extern AppStack
.extern Cstart
.global start

.text
.set noreorder

start:
       la     sp, AppStack
       addiu  sp, 4096
       addiu  sp, -16
goToC:
       la     k0,Cstart
       j      k0
       nop
```

9.4.3 Coldfire-Based Stack Setup

```
.extern       AppStack
.extern       Cstart
.global       start

.text

start:
       move.l #AppStack+(4096-16),%sp
       jsr    Cstart
```

9.4.4 PowerPC-Based Stack Setup

```
#define       sp   1
#define       r0   0
#define       r7   7
```

```
.extern       AppStack
.extern       Cstart
.globl        start

start:
       lis    sp, (AppStack+(4096-16))@h
       addi   sp, sp, (AppStack+(4096-16))@l
       addi   r7, r0, -8
       and    sp, sp, r7
       ba     Cstart
       nop
```

9.5 App #3: Using MicroMonitor's CLI in Application Space

For any application whose hardware has a serial port, it is likely that some kind of command line interface will be needed. MicroMonitor provides a few basic hooks that allows the application to install its own commands, plus make some, all or none of the MicroMonitor commands available to the user. This section will demonstrate that. For these last few applications, we will essentially be using the same make file, memory map file and start.c. Only the new code will be discussed…

```
01: #include "string.h"
02: #include "monlib.h"
03:
04: char *mycmdHelp[] = {
05:     "really doesn't do anything",
06:     "[echo string]",
07:     0,
08: };
09:
10: int
11: mycmdFunc(int argc,char *argv[])
12: {
13:     int  i;
14:
15:     if (argc == 1) {
16:         mon_printf("Error: missing arguments\n");
17:         return(CMD_FAILURE);
18:     }
19:     else {
20:         for(i=1;i<argc;i++)
21:             mon_printf("Arg %d is '%s'\n",i,argv[i]);
22:     }
23:     return(CMD_SUCCESS);
24: }
25:
26: struct monCommand mycmdTbl[] = {
27:     { "mycmd",  mycmdFunc,  mycmdHelp, 0 },
28:     { 0,0,0,0 }
29: };
30:
31: char mycmdUlvltbl[] = { 0 };
32:
33: int
34: main(int argc,char *argv[])
35: {
36:     char line[80];
37:
38:     mon_addcommand(mycmdTbl,mycmdUlvltbl);
39:
40:     while(1) {
41:         mon_printf("MYCLI:");
42:         if (mon_getline(line,sizeof(line),1) > 0)
```

```
43:             if (strcmp(line,"exit") == 0)
44:                 break;
45:             mon_docommand(line,0);
46:
47:     }
48:     mon_printf("Returning control to MicroMonitor...\n");
49:     return(0);
50: }
```

Lines 4-8 establish a help text array that will be used with the new "mycmd" command. The first two strings within this array have specific meaning:

- mycmdHelp[0] is a brief description of the command
- mycmdHelp[1] is the command line usage summary.

All lines after that point are optional, the only requirement is that the final string be a null pointer. This is the same format used by the monitor's built-in commands.

Lines 10-24 make up the command function that is called when the new "mycmd" command is invoked at the command line.

Lines 26-31 establish the command and user level tables (which in this case are made up of only one new command) that is to be appended to the MicroMonitor command table by the call to mon_addcommand() at the top of main().

Line #38 installs the application-based command table into the monitor's command table.

Lines 40-47 make up the application-based command interpreter. This CLI will use "MYCLI" as the user prompt, it will include the line editing and history that is built into the monitor, plus it provides the "exit" command to terminate the loop. One important note here is that some of uMon's commands use the memory space pointed to be $APPRAMBASE for scratch memory. As a result, some commands may be hazardous to the health of the running application if it resides in memory right above $APPRAMBASE.

Run "make app3" and then "make app3dld"... Then at the uMON> prompt, type "app3". At the "MYCLI" prompt, type "help", then "mycmd X Y Z", then "exit". The output of "help" indicates that your CLI is connected with the monitor's CLI. Notice that the commands are part of the monitor's command set, but the new "mycmd" is in the list. Invocation of the "mycmd" command demonstrates that it has been added to the command table, and finally "exit" allows the application to terminate the CLI…

```
uMON>app3
MYCLI:help

Application-Installed Command Set:
mycmd

Micro-Monitor Command Set:
arp         call        cast        cm          dhcp        dis
dm          echo        edit        ether       exit        flash
fm          gdb         gosub       goto        heap        help
?           history     icmp        if          item        mt
mtrace      pm          prof        read        reg         reset
return      set         sleep       sm          strace      ulvl
tftp        tfs         unzip       xmodem      version     date

MYCLI:mycmd 1 2 3
Arg 1 is 'X'
Arg 2 is 'Y'
Arg 3 is 'Z'
MYCLI:exit
Returning control to MicroMonitor...
```

```
Application Exit Status: 0 (0x0)
uMON>
```

If you go back and rerun “app”, try the command line editing. It will work, just as it works at the MicroMonitor command line. If for some reason this is not desirable, then the third argument passed to mon_getline() should be set to zero.

9.5.1 Polled Console IO in a Complex Application

There are a few problems with the above code when considered for use in a more complicated (typical) application:

Assuming the application is an RTOS, it may come with its own serial port driver. This means that at some point during the startup of the RTOS, it initializes its own set of console interface functions and all the code written for that RTOS is supposed to use those functions instead of those provided by the monitor’s API.

Even without an RTOS, if the application does more than just process commands from the console, then the underlying polled serial driver used in mon_getchar() doesn’t release the CPU to do other things in application space.

There are a few options for getting around this problem (with #3 below being the best choice)…

Just don’t use the monitor’s console IO. At first it may not seem like a big deal to lose the monitor’s serial port access; however, realize that if the monitor itself can’t talk to the console port, then the monitor’s CLI is inaccessible from application space. Since the monitor’s CLI has command line editing, shell variables, etc… it’s handy to use; hence, it would be nice to still be able to use it in application space. Plus, if the monitor can’t access the console, then mon_docommand() can’t be used either, since it simply runs a command in the monitor and those commands assume they can read/write at the console using the monitor’s built-in interface functions.

Copy the code from the monitor that supports this stuff into your application and hook it to the application’s console IO functions. This is certainly doable; however, inefficient and just messy.

At runtime, replace the monitor’s putchar/getchar/gotachar functions with application-provided equivalents. The monitor allows the application to install “replacement” functions for mon_getchar(), mon_gotachar() and mon_putchar(). This solves both of the above problems because now you can take advantage of the already-written code in the monitor and at the same time, you overcome the limitation imposed by the polled serial drivers.

Method #3 is mentioned in section 8.2 above, and an example (using VxWorks) is discussed in section 8.5.

9.6 App #4: Hooking Up to TFS in Application Space

MicroMonitor has a TFS (Tiny File System). It is used by the monitor itself to provide a lot of functionality to the system. TFS can also be used by the application; thus, allowing the application to assume the ability to create and read files that will be stored in flash space that is safe from power hits. It is beyond this scope of this section to get into every aspect of the TFS portion of the API, but this example provides a quick start for some of the basic stuff…

```
01: #include "string.h"
02: #include "monlib.h"
03:
04: int
05: printfile(char *filename)
06: {
07:     int tfd;
08:     char c;
09:
10:     tfd = mon_tfsopen(filename,TFS_RDONLY,0);
11:     if (tfd < 0) {
12:         mon_printf("%s: %s\n",filename,(char *)mon_tfsctrl(TFS_ERRMSG,tfd,0));
13:         return(-1);
14:     }
15:     mon_printf("The content of '%s' follows:\n",filename);
16:     while(mon_tfsread(tfd,&c,1) == 1) {
17:         if (c == '\n')
18:             mon_putchar('\r');
```

```
19:         mon_putchar(c);
20:     }
21:
22:     mon_tfsclose(tfd,0);
23:     return(0);
24: }
25:
26: int
27: newfile(char *filename)
28: {
29:     int err;
30:     char *filedata = "This is a file\nwith several lines\nthat can be read\n";
31:
32:     err = mon_tfsadd(filename,0,0,filedata,strlen(filedata));
33:     if (err != TFS_OKAY) {
34:         mon_printf("mon_tfsadd(%s) failed: %s\n",filename,
35:             (char *)mon_tfsctrl(TFS_ERRMSG,err,0));
36:     }
37:     return(err);
38: }
39:
40: int
41: listfiles(void)
42: {
43:     int tot;
44:     TFILE *tfshdr;
45:
46:     tot = 0;
47:     tfshdr = (TFILE *)0;
48:     while((tfshdr = mon_tfsnext(tfshdr)) != 0)
49:         mon_printf("%2d: %s\n",++tot,TFS_NAME(tfshdr));
50:
51:     return(tot);
52: }
53:
54: int
55: main(int argc,char *argv[])
56: {
57:     if (argc != 2) {
58:         mon_printf("Must specify filename\n");
59:         mon_appexit(1);
60:     }
61:
62:     printfile(argv[1]);
63:     newfile("newfile");
64:     listfiles();
65:
66:     mon_printf("Returning control to MicroMonitor...\n");
67:     return(0);
68: }
```

• Listing 25: An Application Using TFS

In this case (referring to Listing 25), the function main() is simply a wrapper for demonstration of three different aspects of interfacing to TFS through an application.

Lines 4-24 (the printfile() function) demonstrate the application's ability to open a file and read from it. This function simply reads one character at a time and dumps it to the console, similar to a UNIX "cat" or DOS "type" command.

Lines 26-38 (the newfile() function) demonstrates the application's ability to quickly and easily create a new file in TFS. This example uses the mon_tfsadd() API function to quickly transfer a buffer in RAM to a file in TFS.

Lines 40-52 (the listfiles() function) demonstrate the application's ability to process the files within TFS by running through the current list of active files.

Once again, run “make app4” and “make app4dld” to build the application and transfer it to the target. Now at the uMON> prompt, just type “app4 monrc” to pass the filename “monrc” to the printfile() function. The output should be similar to the following…

```
uMON>app monrc
The content of 'monrc' follows:
set IPADD 192.168.1.102
set GIPADD 192.168.1.1
set NETMASK 255.255.255.0
 1: monrc
 2: my_first_app
 3: newfile
 4: symtbl
 5: app
Returning control to MicroMonitor...

Application Exit Status: 0 (0x0)
uMON>
```

Obviously the actual output depends on the content of your monrc file. The above listing shows the work of the functions printfile() and listfiles(). Now type “tfs ls” and “tfs cat newfile” to see the new file created thanks to the function newfile()…

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 app                       5964  0xff8c9b5c  E
 monrc                       89  0xff8b035c  e      envsetup
 newfile                     51  0xff8cb30c
 script                      20  0xff8b02ec  e
 symtbl                   30856  0xff8c226c

Total: 5 items listed (36980 bytes).
uMON>tfs cat newfile
This is a file
with several lines
that can be read
uMON>
```

9.7 The “umon_apps/demo” Application

The umon_apps/demo directory is one additional directory that contains basically the same stuff contained in the umon_apps/user_manual directory. The difference is that the umon_apps/demo has only one main.c and one application to build. It can be used as the starting point of an application by simply copying that entire directory to some new application-specific directory and making application-specific modifications. It alsow provides one last demonstration of the use of the monitor’s ability to trace a stack frame after an exception is trapped. Check it out!

9.8 Wrap Up

This section has gone through several examples of hooking an application to the monitor platform below it. The first and second examples showed how to build a very basic application that would link to the monitor. They focused on where the application is mapped, and how it is hooked into MicroMonitor’s API. The last few examples demonstrated some of the functionality provided by the monitor’s API. The next chapter discusses what capabilities the monitor provides to help debug these applications.

Chapter 10 Built-in Diagnostics and Debug

This chapter discusses how MicroMonitor can help a developer debug an application[50]. In the process of doing this, several of the MicroMonitor commands will also be discussed and used. We'll build and install a small example application using the experience gained from Chapter 8 above. Then, we'll walk through some of the things MicroMonitor can do to help in the debugging process. Before getting started in this section, it is important to note that these features are in addition to any debugging capabilities that come with various development environments. The sophistication of some of these MicroMonitor based facilities may seem a bit rudimentary when compared to the capabilities of a JTAG or BDM based development tool suite; however, note that these facilities travel with the target to the field and on customer sites.

10.1 The Application

The following application has no purpose other than to provide a demonstration of some of MicroMonitor's debug facilities. It is "app5" under the umon_apps/user_manual directory and is built and downloaded similar to the application examples of the previous chapter. It initializes a structure, then sits in a loop with a small command line interface processing incoming commands. The point is to simulate a real application that has an accessible command line interface (CLI) Note that we are only looking at the main.c file of the application. There are other files required to build (i.e. makefile, start.c, & link map); however, those files are outside the scope of this application and have been discussed in Chapter 8; hence, they are omitted.

```
1: #include "monlib.h"
2: #include "stddefs.h"
3: #include "genlib.h"
4:
5: #define TRAP()  asm("trap #0")
6:
7: int debug_enabled;
8:
9: struct abc {
10:     long    l;
11:     short   s;
12:     char    c;
13:     char    x;
14:     char    *p;
15: };
16:
17: struct abc abc_s;
18:
19: int value;
20:
21: void
22: syserr(void)
23: {
24:     mon_printf("System Error!\n");
25:     TRAP();
26: }
27:
28: int
29: main(int argc,char *argv[])
30: {
31:     char line[64];
32:
33:     if ((argc == 2) && (strcmp(argv[1],"debug") == 0))
34:         debug_enabled = 1;
35:
```

[50] The majority of these capabilities are CPU and compiler independent, but some portions do require code in the monitor that is specific to the target hardware and the compiler used. That being the case, sections of the following discussion that are not CPU and compiler independent may not be available on all target systems.

```
36:     value = 0;
37:     abc_s.l = 0x12345678;
38:     abc_s.s = 0xBEEF;
39:     abc_s.c = 'Z';
40:     abc_s.x = 'Y';
41:     abc_s.p = "hi mom!";
42:
43:     while(1) {
44:         mon_printf("MYCLI:");
45:         if (mon_getline(line,sizeof(line),1) > 0) {
46:             if (strcmp(line,"exit") == 0) {
47:                 break;
48:             }
49:             else if (strcmp(line,"hi") == 0) {
50:                 mon_printf("Hello\n");
51:                 value ++;
52:             }
53:             else if (strcmp(line,"err") == 0) {
54:                 syserr();
55:             }
56:             else {
57:                 mon_memtrace("Pass '%s' to monitor",line);
58:                 mon_docommand(line,0);
59:             }
60:         }
61:     }
62:     mon_malloc(100);
63:     mon_memtrace("All done!");
64:     return(0);
65: }
66:
67: int
68: testfunc(int arg)
69: {
70:     mon_printf("arg=%d\n",arg);
71:     return(arg+1);
72: }
73:
```

• Listing 26: The Application to be Debugged

Taking a quick walk through Listing 26, main() processes the argument list to check for a “debug” flag. Then a simple structure is initialized, and finally a forever loop is entered to establish a small (CLI) that will allow us to invoke a few commands while the system is up and running to demonstrate various types of symbolic accesses. Notice that the endless loop of the CLI includes an exit path, if the “exit” string is detected. One leg of the CLI allows us to access the syserr() function. This function serves two purposes:

- to simulate an exception (in this case a TRAP) that could be taken by the processor for various reasons;
- to demonstrate the use of a purposely installed exception to catch illegal or unexpected branches in the code.

10.2 Target-Resident Files Used for Symbolic Access

There are two additional files that become useful for the MicroMonitor-assisted debugging process: a symbol table file and a structure definition file (symtbl & structfile respectively). Both of these files must be target-resident to allow MicroMonitor to access symbolic data. Both of these files are target and toolset independent; hence, MicroMonitor’s ability to symbolically access target-resident data works regardless of the CPU (i.e. PPC, ColdFire, 68K etc.) and/or cross-compiler (i.e. GNU, DIAB, GreenHills, etc.).

10.2.1 symtbl

The symbol table for the monitor is a simple file (called “symtbl”) in TFS. Each line in the file is assumed to contain a symbol name followed by a string to replace the symbol name. In all cases here, the

string that replaces the symbol name is a hex address, but this is not a requirement[51]. Refer to Listing 27 for a snippet of a symtbl file[52].

```
start               0x00100008
syserr              0x00100084
main                0x0010009c
abc_s               0x0010393c
debug_enabled       0x00103948
value               0x0010394c
testfunc            0x001001d2
```

• Listing 27: Symbol file

The file is simple ASCII with the symbol being the first white space delimited token and the replacement string being the second white space delimited token. The parsing of each line allows for random amounts of whitespace between the first and second tokens, but requires that each symbol/replacement-string combination be on one line. All cross-compiler toolsets provide the ability to generate some type of readable symbol listing, both "nm" and "objdump" are useful in GNU. Regardless of the tool, the format of the resulting file is likely to vary from one tool chain to the next, so the actual generation of the symtbl file typically requires some manipulation of the toolset-generated symbol file. For those cases where it is just a matter stripping out and/or rearranging columns of white space delimited tokens, the monsym (see section 17.16) tool can be used to help.

10.2.2 structfile

Just as the monitor uses a file to deal with symbolic access, a second file is used to deal with structures. The file "structfile" must be loaded onto the target to tell the monitor what a structure looks like. In this case, the structure definition is almost identical to what it would be in standard C code; however, since it is being used to display data, some extensions are appropriate to allow the monitor to display different objects in different ways. Refer to Listing 28 for a snippet of a typical structure definition file.

```
struct abc {
    long    l;
    short   s;
    char    c;
    char    x;
    char    *p;
};

struct abcx {
    long.x  l;
    short.x s;
    char.c  c;
    char.x  x;
    char.c  *p;
};
```

• Listing 28: Structure definition file

Notice in the listing above that the two structures are almost identical. The difference between the definitions is in the non-C-standard extensions to the various member types. These extensions tell the monitor how to display the member. For a full description of these extensions refer to the cast command in section 15.4 for complete details.

10.2.3 Building the 'symtbl' File:

The steps taken to build the symtbl file vary because it is dependent on the way the compiler/linker toolset builds the symbol file (using some derivative of 'nm' usually). It is usually some type of tabulated output that can be easily parsed with a few shell commands like awk & grep. If these tools are not accessible or you're willing to try out a new tool, MicroMonitor comes with a tool called monsym (refer to section 17.16) that will take

[51] The replacement string can be whatever you want it to be; however, for symbol access, it naturally follows that the replacement string be the address of the symbol.
[52] The content of the symtbl file may show different addresses than your target.

an input file and rearrange the columns to be in the format needed for symtbl. The syntax of each line in the monitor's symtbl file is simply…

<input_string> WHITESPACE <replacement_string>

two whitespace delimited strings per line. The basic output of NM is…

<address> WHITESPACE <section> WHITESPACE <symbol_name>

As a real example of monsym command use, the line…

```
00100008 T start
```

is converted to…

```
start 0x00100008
```

using the following commands (in the makefile for this project)…

```
$(NM) --numeric-sort $(AOUT)  > $(AOUT).sym
monsym -p0x $(AOUT).sym >symtbl_app
cat symtbl_app ../../monitor/symtbl >symtbl
```

Notice the use of the –numeric-sort option to NM. This automatically puts the symbols in ascending (by address) order; this is a requirement for some of the other uses of the symtbl file within MicroMonitor. Also note that the final symtbl file on the host is a concatenation of the symtbl file created for the application as well as the symtbl file for the monitor itself. This isn't necessary, but can be handy for debugging portions of code that use the monitor's API. It then allows you to symbolically access both application and monitor variables.

10.3 Symbolic Debugging

Definition: the ability to look at variables of various types (integer, short, struct, etc.) in the target system without the need to be aware of their actual physical address. This facility is 100% target and toolset independent.

The monitor's ability to do symbolic debugging comes from the command line interface's (CLI) ability to replace "symbols" on the command line with "replacement strings" from a symbol table. Refer to section 3.4 for complete details of the CLI parsing, and note that we will be using this parsing capability to display both runtime and post-mortem symbols in our application. So, install and run the above application on your target system (make; make dld).

10.3.1 Display/Modify Memory Symbolically

With the above capability in mind, we can now type in the following command on the monitor console...

```
MYCLI: dm -4d %value 1
```

and it will be converted by the monitor's CLI to[53]...

```
dm -4d 0x0010394c 1
```

The application-level command interpreter parses the line and doesn't find a match so it passes the entire string to mon_docommnand() (see line 58 of Listing 26), which hands the command over to the monitor's command interpreter. The result is that whatever was stored in the variable "value" as a 4-byte integer will be displayed in decimal format to the user (see dm command (section) for details on output formats); with the user unaware of the physical location of the variable…

```
MYCLI:dm -4d %value 1
```

[53] As always, note that the address may not be exactly the same as your target.

```
0010394c:            0
MYCLI:
```

Now, to verify that we're seeing what we think, issue the command "hi". This invokes the code of lines 50-51 of Listing 26, so the string "Hello" should be printed to the console, plus the content of the variable "value" should be incremented. Then invoke the same "dm" command again and notice that the content of value has incremented…

```
MYCLI:hi
Hello
MYCLI:dm -4d %value 1
0010394c:            1
MYCLI:
```

To modify a location in memory, use the "pm" command (similar in syntax to dm). The following line...

```
pm -4 %value 87
```

will place decimal 87 into the location pointed to by %value, assuming it is a 4-byte integer. Then the "dm" command can be used to verify the change…

```
MYCLI:pm -4 %value 87
MYCLI:dm -4 %value 1
0010394c: 00000057
MYCLI:dm -4d %value 1
0010394c:           87
MYCLI:
```

Notice that two "dm" commands were issued above. One with the "-d" and one without. Clearly, the "-d" option tells the "dm" command to display in decimal (87) format rather than the default hex (00000057) format.

10.3.2 Executing an Embedded Function Symbolically

There are many cases where the easiest way to debug something is to write a function in C and then execute that function from the CLI. The monitor's "call" command provides that capability, and in conjunction with the symbol table, it becomes address independent. The following line...

```
call %testfunc 99
```

will execute the function "testfunc()". The actual address of the function is irrelevant to the user because it is embedded within the symbol table. The function "testfunc()" simply prints the incoming argument, increments it and returns the new value…

```
MYCLI:call %testfunc 99
arg=99
Returned: 100 (0x64)
MYCLI:
```

Notice, in the above call to testfunc, an argument is supplied, and upon return from the function, the call command displays the return value of the call in both hex and decimal. The argument passed was 99 (as was displayed by the mon_printf() function within testfunc()) and the return value was the argument + 1. The important thing to note here is that there was no need to know the address of the function to do this. Refer to the "call" command (section 15.2), for more details.

10.3.3 Symbolically Displaying Raw Memory as a Structure

Finally, for cases where the data to be observed is stored as a structure in memory, the "cast" command is useful. The following line...

```
cast abc %abc_s
```

will display memory starting at the address specified by `%abc_s` as a structure of type "abc" (see cast, section 15.4 below, for more details). This command uses the "structfile" file in TFS for determining what the "abc" structure actually looks like…

```
MYCLI:cast abc %abc_s
struct abc @0x103964:
  long     l: 305419896
  short    s: 48879
  char     c: 90
  char     x: 89
  char     *p: 0x10270d
MYCLI:
```

Alternatively, the structure definition "abcx" (refer to content of structfile in Listing 28) will make the output a bit more readable (depending on the type of data)…

```
MYCLI:cast abcx %abc_s
struct abcx @0x103964:
  long.x   l: 0x12345678
  short.x  s: 0xbeef
  char.c   c: Z
  char.x   x: 0x59
  char.c   *p: "hi mom!"
MYCLI:
```

This second listing demonstrates some of the flexibility of the cast command through the use of "dot" extensions to the various structure members. Once again, in either case, note that there was no need to be aware of the address of the `abc_s` structure.

10.4 Run-time Trace

Definition: The ability to monitor the progress of an executing program. This facility is 100% target and toolset independent.

Despite all the fancy tools that are available, probably the most useful and most easily used debug tool is the "printf()" function. It allows a developer to quickly insert something "visible" into the instruction stream to verify that a particular branch of code is or is not executing. Through the versatility of the variable-argument (i.e. vararg) format string, printf() can dump pointers, strings, integers etc., in a way that is easily read by the developer. Unfortunately, it has its own set of limitations:

- since it is typically implemented on a serial RS-232 port of some kind, it will usually slow down the real-time system dramatically if there are a lot of these printf() calls sprinkled throughout the code.
- in many cases, the underlying driver used to transfer characters to the console is interrupt driven, so insertion of printf() for debug can cause the system to behave in a way that is dramatically different than when printf() is not in the code.
- printf() isn't useful in interrupt handlers simply because it is slow, and in many cases printf() uses interrupts itself.
- in many cases the serial port is used for some other facility; hence, not available for use by debug printf() output.

MicroMonitor has a facility called "memory trace" (refer to the "mtrace" command in section 15.26 below) that overcomes most of these limitations. The facility is a function call in the monitor's API that is coordinated with a command in the monitor's CLI. The function call (mon_memtrace()) is very similar to printf(), but instead of the output going to a serial port, it is tagged and logged to a circular memory buffer previously established by the mtrace command. The command allows the user to configure the location and size of the buffer as well as enable/disable the tracing itself. Then, after enabled, and after the mon_memtrace() function has been called by the running application, the output can dumped to the console or transferred off the target.

At the uMON> prompt type the command “mtrace cfg”. With no additional arguments this command returns the state of the memory trace configuration. At this point it should be unconfigured. The command “mtrace cfg {BASE} {SIZE}” is used to allocate a block of memory in your system to the memory trace facility, so for this example, we’ll assume that it is safe to use memory 1Mg above APPRAMBASE, so issue these two commands: “set MTRACEBUF=$APPRAMBASE+0x100000” followed by “mtrace cfg $MTRACEBUF 0x10000”[54], then type “mtrace on”. These commands configure and enable the memory trace, so that subsequent calls to mon_memtrace() will be logged to the circular buffer allocated at 0x100000 above your target’s APPRAMBASE location. Now, restart the application, type a few commands that will be passed to the monitor, then type “mtrace dump”…

```
uMON> mtrace cfg
Not configured
uMON>set MTRACEBUF=$APPRAMBASE+0x100000
uMON>mtrace cfg $MTRACEBUF 0x10000
uMON>mtrace on
uMON>app
MYCLI:dm -4d %value 1
001039d0:            0
MYCLI:pm -4 %value 99
MYCLI:dm -4 %value 1
001039d0: 00000063
MYCLI:dm -4d %value 1
001039d0:           99
MYCLI:echo hi
hi
MYCLI:mtrace dump

<0001> Pass 'dm -4d %value 1' to monitor
<0002> Pass 'pm -4d %value 99' to monitor
<0003> Pass 'pm -4 %value 99' to monitor
<0004> Pass 'dm -4 %value 1' to monitor
<0005> Pass 'dm -4d %value 1' to monitor
<0006> Pass 'echo hi' to monitor
<0007> Pass 'mtrace dump' to monitor

MYCLI:
```

• Listing 29: Output of the Memory Trace Command

Notice that each time a command is passed to the monitor’s CLI, lines 57&58 of the application (refer to Listing 26) are invoked. Line 57 is a call to mon_memtrace() and line 58 is a call to mon_docommand(). Notice the argument list syntax of the mon_memtrace() call is identical to that of printf(). As a matter of fact, mon_memtrace could be a direct replacement for printf if needed with a simple C-preprocessor line..

```
#define printf mon_memtrace
```

added to the top of a file. The output of “mtrace dump” clearly shows that the basic block of code within lines 57&58 was executed. For this example the path is obvious; however, there are many cases where insertion of mon_memtrace() within questionable code is quite handy, and now since it isn’t actually using a device interface, it can be inserted within an interrupt handler.

10.5 Default Exception Handling

Definition: the ability to process all exceptions taken by the system that are not handled by the application. This facility is very target and toolset dependent.

As a general rule, when developing an application in an embedded system, all exceptions should be give a “handler”. Exception handlers are blocks of code or pointers placed at specific locations within the CPU’s memory space and their job is to deal with exceptions. An exception can occur in a system for several different reasons. Some are expected and some unexpected: external interrupt, trap or system call instructions,

[54] This address is VERY target specific.

execution of an instruction not understood by the CPU, access of a piece of data not properly aligned in memory, etc… The bottom line is that the firmware platform built on the target must be prepared for any exception the CPU may encounter. This doesn't mean it has to be some elaborate mechanism; however, it does have to be accounted for.

MicroMonitor's model for exceptions is to configure all of them as illegal (or unexpected), and in all cases, if an exception is handled by MicroMonitor, the firmware makes a copy of the CPU context (or register set) so that it can be viewed by the user after the exception has transferred control back to the monitor (refer to the "reg" command in section 15.30 below). In addition, the MicroMonitor platform is prepared for the case where the exception occurs in a running system in the field at a customer site (hey, whether you want to admit it or not, this can happen!). In a nutshell, MicroMonitor's exception handling model provides 4 major features:

- **Register Cache:** At the time of the exception, the monitor stores each of the CPU's major registers into a RAM-based array that can be accessed by the "reg" command after the exception has turned over control to the monitor.
- **Violating Address:** At the time of the exception, the monitor attempts to display the address at which the exception occurred. If the symbol table is installed, then it also displays a symbolic equivalent of that address (assuming it is within valid symbol space).
- **Configurable Restart:** The exception handling mechanism can be configured to automatically restart the system or terminate after returning control to the monitor. For a debugging environment, termination to the monitor is what you want (so that you can determine the cause of exception), but in a field situation, you usually want the application to restart as soon as possible. By default, the system automatically restarts; however, this is disabled by setting the shell variable NO_EXCEPTION_RESTART (see section 0 below). The line "`set NO_EXCEPTION_RESTART TRUE`" can be entered at the command line or put into the monrc file. Ideally, during debug, this line should be added to the monrc file.
- **Configurable Action:** In the field, the application usually needs to restart immediately; however, it would be nice if at the time of the exception, some application specific script or program could be run to log exception stats to a file in TFS or to use TFTP to transfer the core to some external system. This can be configured with the shell variable EXCEPTION_SCRIPT (see section 14.26).

Lets apply this to our target. First we'll configure the monitor to not restart after the exception, so type the command "set NO_EXCEPTION_RESTART TRUE" at the MYCLI: prompt. This command will be passed to the monitor's command interpreter (via mon_docommand()) and the internal shell variable will be created. Now type "err". Referring to the code (Listing 26), this will invoke a call to the function "syserr", which has a "man-made" exception. The actual code used here depends on the CPU, so refer to the source code in main5.c under umon_apps/user_manual. After typing "err", the code branches to the syserr() function and the exception is generated…

```
MYCLI:err
System Error!

EXCEPTION: 'Trap #0'
           At 0x100096 (within syserr)

uMON>
```

The exception type and address are very target specific. Note that if the NO_EXCEPTION_RESTART variable was not set, the monitor would have prompted with

```
Press any key to stop auto restart.
```

and if no interaction occurred within about a second, the target would automatically restart; hence, if the application was autobootable it would automatically restart. Notice that as a result of the presence of the symtbl file (discussed in section 10.2.1) the address of the exception as well as its symbolic equivalent are displayed. The exception handler has successfully processed the exception and has returned control to the monitor. A copy of the register set at the time of the exception can be displayed with the "reg" command…

```
uMON>reg
```

```
    PC=0x00100098     SR=0x40802700     A0=0x00102706     A1=0x00103907
    A2=0x0001b84b     A3=0xffffffff     A4=0xffffffff     A5=0xffffffff
    A6=0x001038fc     SP=0x001038f4     D0=0x00000020     D1=0x00000004
    D2=0x00000000     D3=0xfffdffff     D4=0xffffffff     D5=0x00000010
    D6=0xffffffff     D7=0xffffffff
uMON>
```

The actual registers that are displayed vary from CPU to CPU, so this dump is only relevant for the MCF5272 and similar CPU cores.

Now lets try setting the EXCEPTION_SCRIPT shell variable, and re-run the above scenario. Recall that this shell variable configures the monitor to run a script at the time of the exception. This can be useful in cases where you want to catch a bug, but you also need to restart your system if and when the bug occurs. First we need a script. If you ran "make dld" earlier, then the file "except_script" should have been transferred to the target…

```
tfs uname except_ FILE
set SPACE=$APPRAMBASE+0x200000
echo Register Dump: >$SPACE,5000
reg >>
echo Stack Trace: >>
strace >>$FILE
```

• Listing 30: Exception Handling Script

This script example, uses the monitor's CLI redirection capability as discussed in section 3.5 above, plus it demonstrates the "uname" facility within TFS. We're also jumping ahead just a bit by incorporating the "strace" command (discussed next) because this is a perfect place for it. With this script on board[55], issue the commands "set EXCEPTION_SCRIPT except_script" and "set NO_EXCEPTION_RESTART", then restart the application (app). At this point, the application is running with the EXCEPTION_SCRIPT shell variable set to "except_script" and the NO_EXCEPTION_RESTART shell variable cleared (i.e. the system will restart after the exception). Now issue the "err" command at the MYCLI: prompt and watch what happens…

```
uMON>app5
MYCLI:err
System Error!

EXCEPTION: 'Trap #0'
           At 0x100096 (within syserr)

Press any key to stop exception script.
Register Dump:
    PC=0x00100098     SR=0x40802700     A0=0x00102706     A1=0x00103907
    A2=0x0001b84b     A3=0xffffffff     A4=0xffffffff     A5=0xffffffff
    A6=0x001038fc     SP=0x001038f4     D0=0x00000020     D1=0x00000004
    D2=0x00000000     D3=0xfffdffff     D4=0xffffffff     D5=0x00000010
    D6=0xffffffff     D7=0xffffffff
Stack Trace:
   0x00100098: syserr() + 0x14
   0x00100192: main() + 0xf6
   0x0010006e: start() + 0x66
Press any key to stop auto restart.
TFS Scanning //FLASH/...
MICRO MONITOR
CPU: MCF5272
Platform: Cogent CSB360 MCF5272 SBC
Built: Jan 25,2004 @ 18:58:19
Monitor RAM: 0x000400-0x01b84c
Application RAM Base: 0x01c000
MAC: 00:60:1d:02:0b:87
IP: 192.168.1.102
```

[55] Don't forget that the script is executable, so when installing it in TFS, include the 'e' flag.

```
uMON>
```

• **Listing 31: Exception Script Execution with Restart**

So what happened? Referring to Listing 31, first the normal exception handler ran, then the automatic execution of the except_script ran, then instead of just seeing a uMON> prompt, the monitor went through a complete reset (because the NO_EXCEPTION_RESTART variable was cleared). If the "app" program that we have on the system was configured to be autobootable, then the application would have restarted automatically. Now check out what's in TFS. Type "tfs ls", then "tfs cat except_0"…

```
uMON>tfs ls
 Name                       Size   Location   Flags  Info
 app                       10964  0xff96a76c  E
 except_0                    447  0xff9755ec
 except_script               101  0xff97551c  e
 monrc                        92  0xff9405ac  e
 structfile                  157  0xff95576c
 symtbl                    31801  0xff96d29c

Total: 6 items listed (43562 bytes).
uMON>tfs cat except_0
Register Dump:
    PC=0x00100098     SR=0x40802700     A0=0x00102706     A1=0x00103907
    A2=0x0001b84b     A3=0xffffffff     A4=0xffffffff     A5=0xffffffff
    A6=0x001038fc     SP=0x001038f4     D0=0x00000020     D1=0x00000004
    D2=0x00000000     D3=0xfffdffff     D4=0xffffffff     D5=0x00000010
    D6=0xffffffff     D7=0xffffffff
Stack Trace:
   0x00100098: syserr() + 0x14
   0x00100192: main() + 0xf6
   0x0010006e: start() + 0x66
uMON>
```

• **Listing 32: File Generated by Exception Script**

Once again, realize that the actual register dump is CPU/target specific. Referring back to the script of Listing 30, the filename "except_0" was derived by TFS with the subcommand "uname". This command is used to build a filename (with a fixed prefix and derived suffix) of a file that does not exist in TFS space. Then the "reg" and "strace" output was redirected to that file. If the exception occurred again, the file "except_1" would have been generated (and so on). The except_script could have also contained a tftp command that would transfer the core to some TFTP server local to the embedded system. What gets put in this script is entirely system-dependent, but provides a lot of versatility for catching problems that occur when nobody's watching.

10.6 Context-Sensitive Stack Trace

Definition: the ability to determine how the system reached a certain point in the code (function nesting) for both single and multi-threaded applications. This facility is very target and toolset dependent.

We've mentioned this in the previous section. What do you do if you hit a piece of code (or take an exception) and want to know how you got there? A stack trace capability is invaluable in cases like this. MicroMonitor has a stack trace capability that allows the user to step back through the function nesting to see each of the functions that have been called to reach the current context. This has two immediately handy uses:

- If an exception occurs as a result of some kind of programming error (bad instruction, etc.), in many cases the strace command can be used to determine how the code got there. Note that this depends on just how bad the error corrupted the system; hence, unfortunately strace is not always going to give you what you want here.
- The user can insert a "man-made" exception into an error leg of the code and upon taking that exception the user can quickly determine how the code got there. This is what we used the syserr() function for in the application of Listing 26.

The output of this command shows you exactly what function nesting caused the code to branch to syserr(). Referring to a snippet of output shown in Listing 32, the strace command generates the following…

```
uMON>strace
   0x00100098: syserr() + 0x14
   0x00100192: main() + 0xf6
   0x0010006e: start() + 0x66
uMON>
```

For this example, the path was pretty obvious; however, imagine you have a multi-person project with tens of thousands of lines of code and syserr() is called from one of several hundred different points. This immediately becomes useful!

The output of strace includes the address, function name and offset into the function. Note that because of the symbol table being on-board, the function name is included in the output[56]. For strace to be able to do this, the symbols in the symbol table must be ordered by increasing address. For example, in the above listing, the address 0x00100192 is not the start of main(), its somewhere within main(); hence, strace needs to be able to scan through the list of addresses, find the first address that is greater than the one it is trying to symbolically convert, then back up one; hence, the address list is treated as a list of ranges in this case because 0x00100192 is somewhere between the address of main and the address of the next function after main in the symbol table. This numerical ordering can be done with the NM command or with the –S option of the monsym tool (section 17.16).

This example of the strace command is based on a single threaded application. The strace command can support a multi-threaded environment in which case several tasks are running, each of which have their own thread of execution. The strace command has options that allow the user to specify where to get the stack frame from; hence, if the appropriate details of the RTOS are available at the time of a crash like this, then the current function nesting of each task could be dumped. Refer to the strace command (section 15.36 below), for more details.

10.7 Post Mortem Analysis

Now, assuming you are walking through this section sequentially, your target has just returned control from application to monitor. In a real system, this could be due to some bug, or an actual exit of the application. Bottom line, your application has terminated or crashed. Since the monitor is "underneath" the application, termination of the application results in a restart of the monitor. The result is that you now have the ability to investigate what is commonly referred to as the "core" of the program. The "core" is simply the state of the application's memory space after some catastrophic event. You've probably heard the term "core dump". On Unix systems, if an application crashes, the operating system takes over and creates a file called "core" that can be observed by a debugger after the event… very useful. For MicroMonitor based applications, a similar capability is available. Getting back to our scenario where our application has terminated or crashed and the monitor has taken over, we essentially have a "core" in the RAM of the target. All of the above facilities can be used to analyze the RAM-resident core, plus, if necessary, the content of the target-resident RAM could be transferred (TFTP or Xmodem) to a host system for later analysis.

```
uMON>dm -4d %value 1
00103974:           87

uMON>mtrace dump

<0001> Pass 'pm -4 %value 87' to monitor

uMON>cast abcx %abc_s
struct abcx @0x103964:
  long.x   l: 0x12345678
  short.x  s: 0xbeef
  char.c   c: Z
  char.x   x: 0x59
  char.c   *p: "hi mom!"
```

[56] If the symbol table was not present, then only the addresses would be shown.

```
uMON>
```

Notice in the above listing that re-running the commands that were run while the application was running gives the same result now that the application has been terminated. The only difference is that now the uMON> prompt is seen instead of the application's "MYCLI:" prompt (indicating that the monitor is in control).

10.8 Profiling Your Application

Ideally, all functions in an application should be optimized as much as possible. Realistically, the functions that are most heavily used by the application should be the ones that are most heavily optimized. The purpose behind "profiling" an application is to determine what functions (or tasks) are active the most. This provides the developer with pointers to the code that should be worked on the hardest. This section discusses how MicroMonitor can help a developer profile an application.

The "prof" command is used to configure the runtime portion of the profiler and to dump the statistics gathered by the profiler. The runtime portion of this is handled through a call to mon_profiler(). This monitor API function is called with a pointer to a monprof structure...

```
struct monprof {
    unsigned long type;
    unsigned long pc;
    unsigned long tid;
}
```

Refer to monprof.h (part of the MicroMonitor common source) for the latest information. Currently there are three different modes of operation (bitfields in the 'type' member) supported by this profiler: MONPROF_FUNCLOG, MONPROF_TIDLOG and MONPROF_PCLOG[57]. Each of them have their value and depend on the CPU and facilities available in the target hardware. The application simply includes monprof.h as a header in the file that contains some high-priority interrupt (preferably the system tick) that can call mon_profiler(). It must load the 'pc' entry of the monprof structure with the address that was interrupted by this interrupt handler, and the 'tid' entry with the task id of the running task.

Note that MONPROF_FUNCLOG and MONPROF_PCLOG need the 'pc' entry and MONPROF_TIDLOG needs the 'tid' entry; hence, both the pc and tid entries may not be required information.

10.8.1 Task ID Statistics

Definition: the ability to determine what tasks are statistically the most active.

The prof command is used to initialize the profiler so that it knows how many different TID values to keep track of. Each time mon_profiler is called the TID value is compared to all tid values already logged (binary search), if a match is found, then that TID count is incremented; if no match is found then the new TID value is inserted into the list. The list is kept sorted so that the TID search is kept efficient. Following is an example code snippet that would be part of the application's system tick handler or some other high-level interrupt...

```
struct monprof mp;

mp.type = MONPROF_TIDLOG;
mp.tid = getCurrentTaskId();     /* application/RTOS specific */
mon_profiler(&mp);
```

10.8.2 Function Statistics

Definition: the ability to determine what functions are statistically the most active.

[57] The term "PC" is used heavily in this section. In this context, "PC" refers to "Program Counter" (not personal computer). The actual name is dependent on the CPU, it is also commonly referred to as the "Instruction Pointer" or "Instruction Address".

The prof command is used to initialize the profiler based on the content of the "symtbl" file. This file is assumed to have its entries in ascending address order, so the profiler uses the addresses as symbolic ranges into which each interrupted PC will fall. For example, if the content of symtbl was:

```
main  0x123000
func  0x123804
func1 0x124008
```

then any interrupted PC value between 0x123000 and 0x123803 would be considered a hit to the function main(). This profiling method uses the content of symtbl, so it is wise to remove all but the function addresses from symtbl for this profiling method. Optimizing symtbl provides two benefits: less RAM is needed to store this data and less time is taken in the interrupt handler to determine which symbolic range the PC has fallen within. At runtime, each call to mon_profiler() results in a binary search through the list of function symbol addresses looking for a match between the incoming PC and one of the symbol address ranges.

Following is an example code snippet that would be part of the application's system tick handler or some other high-level interrupt...

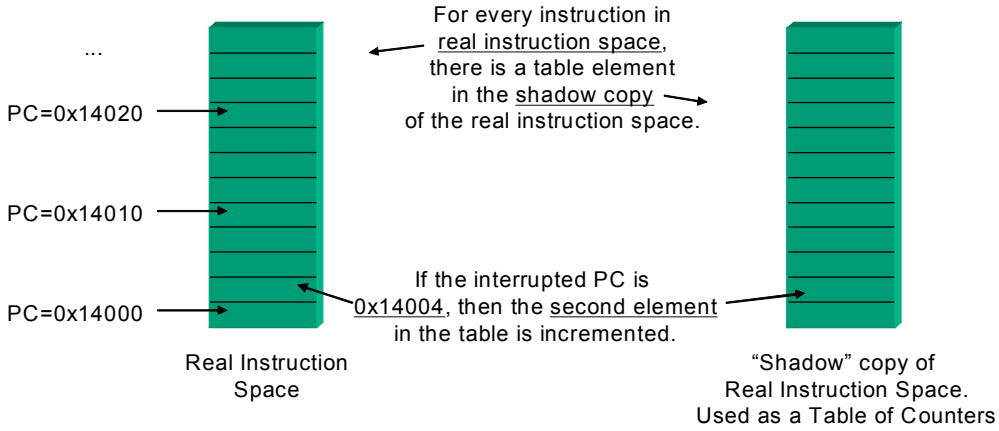
```
struct monprof mp;

mp.type = MONPROF_FUNCLOG;
mp.pc = getInterruptedPC();     /* application/RTOS specific */
mon_profiler(&mp);
```

10.8.3 PC Statistics

Definition: the ability to determine what lines of code are statistically the most active.

The prof command is used to initialize the profiler based on the size of the application's instruction space (size of .text section). The runtime profiler assumes that all instructions are at least the size of an integer and that an array of space equal to the size of the instruction space (referred to here as the shadow text space) is available for this profiling[58]. At the call to mon_profiler() this mechanism simply takes the interrupted PC and computes an offset into the shadow text space based on the PC value and the start of the real .text space. The location of the offset within the shadow space is incremented; hence, this provides the ability to log each instruction that is interrupted at runtime. If the memory space is available, then this is a very efficient profiling alternative; the run-time profiler is simply incrementing a location in a table based on the interrupted PC and some already computed offset between the actual text space and the shadow text space used to log the data. Refer to the diagram below...

[58] The integer-wide instruction width is required because this mechanism assumes that each instruction location in the shadow .text space can be incremented as an integer. The code could be modified so that the increment is only character-wide; however, then the counters can easily wrap.



As the system tick interrupts various locations in real instruction space, the call to mon_profiler() with that address will increment the corresponding location in the "shadow" copy of instruction space.

PC-Level Statistical Profiling

Following is an example code snippet that would be part of the application's system tick handler or some other high-level interrupt...

```
struct monprof mp;

mp.type = MONPROF_PCLOG;
mp.pc = getInterruptedPC();     /* this is application/RTOS specific */
mon_profiler(&mp);
```

10.8.4 Example Profiling Session

Let's assume we have an application called "threads". Its an RTOS-based program with three tasks, and a memory map as is shown by the output of "tfs -v ld threads" below. The function "sysTick()" is the system tick handler (high level periodic interrupt), the function getInterruptedPC() will return the address of the instruction that was running just as the interrupt occurred and the function getCurrentTid() will return the ID of the task that was running prior to the interrupt occurring.

```
uMON>tfs -v ld threads
.text     : copy   14200 bytes from 0xf01b308c to 0x00030000
.data     : copy     264 bytes from 0xf01b6804 to 0x00033778
.rodata   : copy     208 bytes from 0xf01b690c to 0x00033880
.got      : copy      16 bytes from 0xf01b69dc to 0x00033950
.bss      : set      204 bytes  at  0x00033960 to 0x00
.sbss     : set       12 bytes  at  0x00033a2c to 0x00
```

We are running on a CPU that has fixed size instructions (4 bytes per) and enough spare memory to allocate a block of RAM equal in size to the .text section of our application (14200 bytes). In the application we insert the following code into the sysTick() function:

```
struct monprof mp;

mp.type = MONPROF_PCLOG | MONPROF_TIDLOG;
mp.pc = getInterruptedPC();
mp.tid = getCurrentTid();
```

```
mon_profiler(&mp);
```

Prior to running the application, we must configure the profiler code in the monitor. This is done with the "prof" command. There are several steps:

Initialize (clear) the profiling statistics and control structures

```
prof init
```

Configure the profiling mechanisms to be used. In this case we are using the PC and TID logging. For this example we are running an application with 3 task ids and a text section of size 14200 bytes starting at 0x30000...

```
prof tidcfg 3
prof pccfg 4 0x30000 14200
```

Enabling profiling. Without this, the mon_profiler() function would simply return with no logging performed.

```
prof on
```

At this point, the application can be started and statistics will be gathered. At completion of the application profiling run, the prof command is used to dump the results of the statistics...

```
prof show
```

The output of prof show is something like this...

```
01: FuncCount Cfg: tbl: 0x00000000, size: 0x0
02: TidCount  Cfg: tbl: 0x00022000, size: 0x3
03: PCCount   Cfg: tbl: 0x00022018, size: 0x3778
04:
05: TID_PROF stats:
06:   00000002    :  1
07:   01114000    :  3
08:   01114800    :  2
09:
10: PC_PROF stats:
11:   00030804    :  1
12:   00031214    :  1
13:
14: 4 pc out-of-range hits
15: 6 total profiler calls
```

The output above shows the results of the profiling as a result of running the application with profiling enabled (normally, there would be no line numbers). Lines 1-3 show the configuration of the profiler (indicating that the function counting portion of the profiler was not active). Lines 5-8 show the results of the task-id profiling… three tasks each with some TID hit count. Lines 10-12 show the instruction addresses that were non zero in the shadow instruction space (normally there would be more than just two but for this example, the output is limited). Finally, lines 14-15 show the PC values that were outside the range of the text space being profiled and the total number of profiler calls. The output will obviously vary, and since this is not an application that we are actually running on our target, we can't dig too deep here. The point to be noted is that there are three profiling techniques that the monitor provides to an application and depending on the system, various combinations have applicability.

Note that this technique of profiling adds overhead to the system being profiled, so that obviously has to be considered.

10.9 Heap Corruption & Memory Leak Detection

Generally speaking, dynamic memory allocation is not something that should be used in an embedded system. This is because over time heap fragmentation can cause calls to malloc/realloc that were successful in

day #1 of project deployment to start failing in day #100 of project deployment. However, despite the very valid warnings of the use of malloc in embedded systems, if it is used with discipline then under certain circumstances it is a valuable part of the application library. The monitor has it's own malloc-like set of API calls and when used, the "heap" command can provide some insight into the state of the allocated memory. Also, internally, each time a malloc/realloc/free function is called, the entire state of the heap is tested. This does slow down the runtime performance; however, it almost guarantees that you will detect corruption when the heap is accessed through its API.

```
uMON>heap
Heap summary:
  Malloc/realloc/free calls:  39/0/1
  Malloc/free totals: 1848/8
  High-water level:   1840
  Malloc failures:    0
  Bytes overhead:     624
  Bytes currently allocated:   1840
  Bytes free on current heap:  63072
  Bytes left in allocation pool:  0
uMON>
```

• Listing 33: Output of heap command

The text of Listing 33 shows the output of the heap command. This basic set of statistics provides the user with enough information to detect memory leaks. Particularly, the summary of the number of malloc/realloc/free calls gives the user a quick idea of the dynamics of the system. The total space allocated and freed gives a good hint as to whether or not there is a leak, plus the "High-water level" shows the maximum amount of memory that was ever allocated at a given point in time. Since "heap" is a standard, target-independent MicroMonitor command, it can be part of the application through use of the mon_docommand() API; hence, it is useable during the runtime of the application.

As an example of the basic capability of this facility in the monitor, run the heap command, then run the application above and type "exit", then run the heap command again. Notice that the call to mon_malloc() (line 62 of Listing 26) is detected by the fact that the number of malloc calls has incremented. This immediately lets the user know that there is a leak…

```
uMON>heap
Heap summary:
  Malloc/realloc/free calls:  39/0/1
  Malloc/free totals: 1848/8
  High-water level:   1840
  Malloc failures:    0
  Bytes overhead:     2028
  Bytes currently allocated:   1840
  Bytes free on current heap:  61668
  Bytes left in allocation pool:  0
uMON>app
MYCLI:exit

Application Exit Status: 0 (0x0)
uMON>heap
Heap summary:
  Malloc/realloc/free calls:  40/0/1
  Malloc/free totals: 3296/1288
  High-water level:   2008
  Malloc failures:    0
  Bytes overhead:     2132
  Bytes currently allocated:   2008
  Bytes free on current heap:  61396
  Bytes left in allocation pool:  0
uMON>
```

Notice the bold/underlined counts above. The number of malloc/realloc/free calls prior to running the application was 39/0/1 and the number after the application was 40/0/1. This shows you that the number of calls to malloc increased at a higher rate than the number of calls to free; hence, an indication that there is some leak in the program. Note that all of the other numbers will vary depending on the current state of your target, so it is important to realize that this data must be studied and applied to each application; nevertheless, it is a simple way to quickly retrieve memory leak information.

A verbose dump (include the –v option) of each of the allocation blocks can also be seen. This allows the user to get an idea of what is actually in the allocated blocks…

```
uMON>heap -v
        addr     size free?    mptr        nxt        prv      ascii@addr
  0: 0x00000414  1280   n   0x00000404 0x00000914 0x00000000  ................
  1: 0x00000924    68   n   0x00000914 0x00000968 0x00000404  ......U...T....P
  2: 0x00000978    16   n   0x00000968 0x00000988 0x00000914  ................
  3: 0x00000998     8   n   0x00000988 0x000009a0 0x00000968  PROMPT..
  4: 0x000009b0     8   n   0x000009a0 0x000009b8 0x00000988  uMON>...
  5: 0x000009c8    16   n   0x000009b8 0x000009d8 0x000009a0  ................
  6: 0x000009e8    12   n   0x000009d8 0x000009f4 0x000009b8  APPRAMBASE..
  7: 0x00000a04     8   n   0x000009f4 0x00000a0c 0x000009d8  0x1c000.
  8: 0x00000a1c    16   n   0x00000a0c 0x00000a2c 0x000009f4  ...X...<.......t
  9: 0x00000a3c    12   n   0x00000a2c 0x00000a48 0x00000a0c  BOOTROMBASE.
 10: 0x00000a58    12   n   0x00000a48 0x00000a64 0x00000a2c  0xff800000..
 11: 0x00000a74    16   n   0x00000a64 0x00000a84 0x00000a48  ................
 12: 0x00000a94    12   n   0x00000a84 0x00000aa0 0x00000a64  PLATFORM....
 13: 0x00000ab0    28   n   0x00000aa0 0x00000acc 0x00000a84  Cogent CSB360 MC
 14: 0x00000adc    16   n   0x00000acc 0x00000aec 0x00000aa0  ...............D
 15: 0x00000afc    16   n   0x00000aec 0x00000b0c 0x00000acc  MONITORBUILT....
 16: 0x00000b1c    24   n   0x00000b0c 0x00000b34 0x00000aec  Feb_10,2004@06:5
 17: 0x00000b44    16   n   0x00000b34 0x00000b54 0x00000b0c  ...|...d........
 18: 0x00000b64     8   n   0x00000b54 0x00000b6c 0x00000b34  CMDSTAT.
 19: 0x00000b7c     8   n   0x00000b6c 0x00000b84 0x00000b54  PASS....
 20: 0x00000b94    16   n   0x00000b84 0x00000ba4 0x00000b6c  ................
 21: 0x00000bb4     8   n   0x00000ba4 0x00000bbc 0x00000b84  IPADD...
 22: 0x00000bcc     8   n   0x00000bbc 0x00000bd4 0x00000ba4  NETMASK.
 23: 0x00000be4    16   n   0x00000bd4 0x00000bf4 0x00000bbc  ... ...........8
 24: 0x00000c04    12   n   0x00000bf4 0x00000c10 0x00000bd4  CONSOLEBAUD.
 25: 0x00000c20     8   n   0x00000c10 0x00000c28 0x00000bf4  38400...
 26: 0x00000c38    16   n   0x00000c28 0x00000c48 0x00000c10  ...t...X........
 27: 0x00000c58    12   n   0x00000c48 0x00000c64 0x00000c28  ETHERADD....
 28: 0x00000c74    20   n   0x00000c64 0x00000c88 0x00000c48  00:60:1d:02:0b:8
 29: 0x00000c98    16   n   0x00000c88 0x00000ca8 0x00000c64  192.168.1.102...
 30: 0x00000cb8    16   n   0x00000ca8 0x00000cc8 0x00000c88  ................
 31: 0x00000cd8    16   n   0x00000cc8 0x00000ce8 0x00000ca8  255.255.255.0...
 32: 0x00000cf8    16   n   0x00000ce8 0x00000d08 0x00000cc8  ...0...........L
 33: 0x00000d18     8   n   0x00000d08 0x00000d20 0x00000ce8  GIPADD..
 34: 0x00000d30    12   n   0x00000d20 0x00000d3c 0x00000d08  192.168.1.1.
 35: 0x00000d4c    16   n   0x00000d3c 0x00000d5c 0x00000d20  .......l........
 36: 0x00000d6c    16   n   0x00000d5c 0x00000d7c 0x00000d3c  DHCPLEASETIME...
 37: 0x00000d8c     8   n   0x00000d7c 0x00000d94 0x00000d5c  0x15180.
 38: 0x00000da4 63072   y   0x00000d94 0x00000000 0x00000d7c  ................

  Malloc/realloc/free calls:  39/0/1
  Malloc/free totals: 1848/8
  High-water level:   1840
  Malloc failures:    0
  Bytes overhead:     624
  Bytes currently allocated:   1840
  Bytes free on current heap:  63072
  Bytes left in allocation pool:  0
uMON>
```

• Listing 34: Output of heap -v

The above dump (Listing 34) shows the content of the internals of the heap, which when necessary, can be useful for determining the level of fragmentation (if any) that is occurring over time. In addition, if the monitor is built with MALLOC_DEBUG enabled (refer to common/monitor/malloc.c for details), then the heap statistics will include the file and line number of the access, so finding the source of a memory leak can be quite easy with this built in to the monitor's memory manager. The above verbose output would include the file & line number of the allocation call, similar to the following…

```
…
34: 0x0000121c    12   n   0x000011e8 0x00001228 0x000011ac  192.168.1.1.
     ../../../common/monitor/env.c 295
35: 0x0000125c    16   n   0x00001228 0x0000126c 0x000011e8  ................
     ../../../common/monitor/env.c 287
36: 0x000012a0    16   n   0x0000126c 0x000012b0 0x00001228  DHCPLEASETIME...
     ../../../common/monitor/env.c 291
…
```

In the above output, the filenames are actually MicroMonitor source files; however, this can be inherited by the application if mon_malloc(), mon_calloc() and mon_realloc() are redefined to include __FILE__ and __LINE__ as the final two arguments to the call[59]. Note that the only requirement here is that the monitor is specifically rebuilt to use this modified API, refer to common/monitor/monlib.h and common/monitor/monlib.c for the changes based on the definition of MALLOC_DEBUG. Refer to malloc.c for notes on how to convert the monitor to this API. In a nutshell, simply include the file "mallocdebug.h" in the monitor's config.h file and rebuild.

10.10 Breakpoints and Single Stepping… NOT

This entire chapter discussed the facilities that MicroMonitor provides for debug and diagnosis of an embedded application; however, there was no mention of what would appear to be the most obvious debug capability of all… breakpoints and single stepping.

When I first wrote MicroMonitor it wasn't even called MicroMonitor; it was just a boot monitor that was used on a 68K based project. It did all of the normal stuff a monitor would do… display and modify memory, download code, jump into code (etc…), plus it provided single stepping and breakpoint capability. That, together with knowledge of the operating system that runs on top of it, plus some help from a host-based source level debugger provided a really nice environment for development and debug of an embedded system application. So why not carry that model through for MicroMonitor in general… In a nutshell, it's really hard! The basic capability of dealing with breakpoints and single stepping is just the tip of the iceberg. All kinds of issues make this kind of monitor-based debugging very tedious to implement; hence, impractical to have as part of a generic embedded system boot platform. For debugging a simple, single-threaded application it's handy to have; however, how many embedded system applications are "simple, single-threaded applications" nowadays? This is where it gets tricky, and ultimately requires unique consideration for each and every target implementation, even for those with the same CPU. Here are just a few of the things that need to be considered…

- Breakpoints and single stepping use the CPU's exception table. This means that the application has no choice but to share the exception handlers with the monitor. This isn't so bad, since it's not unusual for the monitor to "own" the exceptions that are not used by the application. On the other hand, if the application wants to own the exception table itself, there is an immediate conflict.
- When the monitor runs, it has its own "view" of the target hardware and the state of that hardware. Drivers that interface to the serial port and Ethernet port can be written to meet the needs of the monitor. Now put an application on top that has its own driver model that it installs. So far so good, since the application is running, it's ok for it to take over the peripherals. But, if we supposedly have the monitor in control of breakpoints, then the monitor takes over when a breakpoint occurs. Hmmm… wait a minute, if the monitor takes over, then it has to re-install its view of the drivers; but what happens when it's time to say "continue"

[59] The __FILE__ and __LINE__ macros are common in most compilers. They are processed at compile time such that __FILE__ is the full path of the source file, and __LINE__ is the line number of the file.

or “resume” from the point at which the breakpoint occurred? Now the monitor how has to uninstall its drivers and reinstall the drivers used by the application. This is essentially impossible without an intimate relationship between monitor and application.

- What happens to “time” when in a breakpoint? If the breakpoint occurs and a transition from application to monitor is done, what happens to those interrupts that occur during the time when the monitor is supposedly holding the application off?

10.11 GDB Interface

MicroMonitor can be configured with a gdb server running on Ethernet (a preliminary serial interface is also available, but untested as of this writing). The gdb stubs in uMon1.0 allow the user to connect to the target via the gdb command:

```
target remote udp:192.168.1.110:1234
```

This assumes the target’s IP address is 192.168.1.110, and uMon1.0 uses port 1234 for the gdb connection[60]. For uMon1.0, the gdb stubs provide the ability to download and run the application, then upon termination of the application, variables (or the ‘core’) can be analyzed. There are no breakpoint and/or single stepping capabilities at this time; however, that does not preclude the application from providing them on top of MicroMonitor.

10.12 Wrap-Up

MicroMonitor’s intent is to be generic. The majority of the facilities discussed in this section are 100% independent of the target, CPU and RTOS. The breakpoint/single-step capability is almost a direct contradiction to this philosophy. While there is no question that implementing this for a single application can be quite useful, re-implementing it for every successive application can be quite tedious; hence, impractical. Plus, in most cases, when a commercial RTOS is used, it has it’s own concept of a debugger, it is usually built in to the operating system itself, and just comes with the package. The debug features available with MicroMonitor just add to that.

[60] The value of 1234 is the default and can be overridden in uMon by setting the GDBPORT shell variable to the desired port number.

Chapter 11 Porting to a New Target

One of the early goals of the MicroMonitor design was to make it simple to port to new targets. Over the years, MicroMonitor has been ported to x86, 68K/ColdFire, Hitachi-SH, ARM/XSCALE, PowerPC, MIPS, MicroBlaze, NIOS and Blackfin processor families on a variety of different target platforms. The only requirement that MicroMonitor puts on the target is that it contains on-board flash & RAM (enough to deal with MicroMonitor's footprint), and either a serial or Ethernet port for target-to-host communication. In general, MicroMonitor is not intended for small microcontroller-based systems; however with the recent jump in memory map size in some of these chips, MicroMonitor on a microcontroller is probably not out of the question.

This chapter will walk through an entire porting process starting with a template umon port as the target source that will eventually turn into a target-specific port that includes a flash file system, serial port console, Ethernet connectivity and all of the features supported by the monitor firmware. Prior to starting the port, a few issues need to be made clear and if the hardware is not yet designed, a few decisions need to be made.

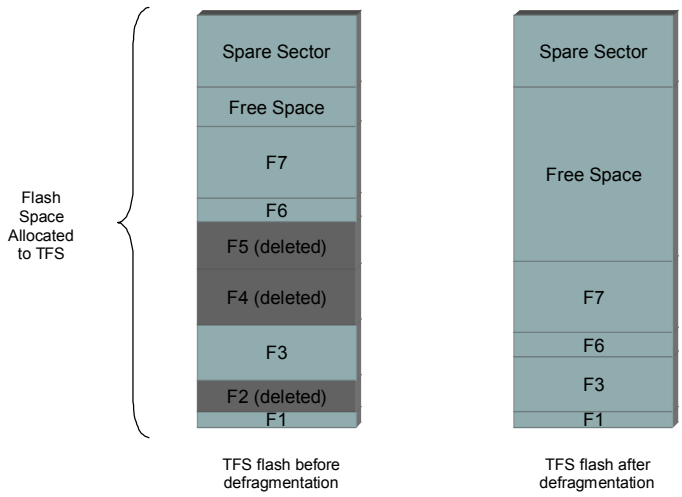
11.1 First Things First

Before beginning a port, and if possible, before completing the hardware design, make sure you are aware of what the monitor needs and provides. It assumes a CPU with linear address space (no bank switching, and if x86, then use a memory model that eliminates the need for C-code to deal with segmentation). To use TFS, an obvious assumption is on-board flash. TFS assumes that the flash is accessible directly on the address/data bus, there is no IDE or I2C (or any other) interface between the memory allocated to TFS and the CPU (future versions of TFS may support these interfaces).

11.1.1 Flash Life Expectancy

Make sure you are aware of the flash life expectancy issues with TFS and based on the features that you decide to use within the monitor platform, make sure you have the memory space to allocate for it. It is important to be aware of the fact that the underlying technology (flash) has a limited number of erase cycles. Current flash devices typically support 100,000 to 1,000,000 erases per sector. Applications will use TFS in different ways, so it is impossible to draw any general conclusions here with regard to how long the flash will last in a system using TFS. This section will; however, discuss the way TFS uses the flash so that developers can determine flash lifetime based on their application's intended use of the file system.

The underlying technique used by TFS to store files in flash is extremely simple. The data within each file is stored in contiguous memory space in the flash. The files are stored as a linked list of structures, and as each file is added, it is appended to the end of the linked list. A file is deleted by simply clearing a bit in the file's header to indicate that it is "dead". After that, if a file with the same name is added, it is appended to the end of the list. Eventually this process reaches the end of the physical flash space and the storage area must be cleaned up (or defragmented). The idea behind the defragmentation is quite simple…



• **Figure 5: File Shift as a Result of Defragmentation**

Referring to Figure 5, the space used by the dead files (shaded in the left map) is erased and all valid files are shifted downward to reside back-to-back in that space. The goal of this is to create free space at the end of the linked list for new files to be added. The complicated part here is making this defragmentation interruptible by a power hit or reset. Then, when the target restarts it can pick up from where it left off in the defragmentation process so that the file system is not corrupted. This requires a “spare” sector for temporary storage of data while the flash is being defragmented. How this defragmentation "wears out" the sectors depends on what is being done. The defragmentation process tries to be as "sector friendly" as possible, but doesn't do anything very sophisticated to do this. The spare sector is usually going to be the sector that is hit the hardest, but even this depends on the organization of the files and the frequency of their deletion. In an absolute worst case defragmentation, each sector in the file storage area, is copied to the spare sector while that sector is defragmented. This means, for the worst case, if you have N sectors dedicated to TFS for file storage, the spare sector is written/erased N times per defragmentation. Fortunately, this worst case is much worse than typical (but should not be ignored). Usually a set of files is stored in TFS and a subset of those files is updated periodically. When the defragmentation process takes place, it shifts all existing files to the bottom of TFS storage space; hence all the files that are not updated often tend to be at the base of the flash and the files that are updated often are toward the end of the flash. When defragmentation starts, it does not touch the spare sector until it finds the first dead file; hence, the worst-case scenario mentioned above is immediately reduced based on the size of the file space that is not updated often. Plus, if the contiguous space taken up by a dead file (or files) exceeds the size of a sector in TFS, then the spare will not be needed for defrag of that sector.

Since its tough to make a general statement about how long you can expect your flash to "live", lets take an example of worst case just so that the numbers are put into perspective... Assume you have a 64 sector TFS flash space and every defrag requires every sector to be copied to the spare prior to updating the sector. This means that the spare sector is erased about 66 times per defragmentation. Assuming a defragmentation is done once a week, and assuming you have a flash device with a life expectancy of 100,000 erase cycles (many support 1,000,000 now), then your device will last 29 years ((100,000/66)/52) before it reaches the 100,000 erase count. All of the criteria in this example is conservative[61], so in this case, 29 years is an absolute minimum life span. The bottom line is this: regardless of TFS’s attempts to be “sector-friendly”, be aware that TFS makes no attempt to do sector wear-leveling. If the file system you need is to be used heavily enough to

[61] Plus, the 100,000 or 1,000,000 number advertised by the flash device manufacturer is a minimum.

require wear-leveling, then buy one! Wear leveling adds a great deal of complexity to the underlying implementation and is beyond the scope or purpose of TFS. The goal of TFS integrated into MicroMonitor is to provide a extensible platform that satisfies the needs of most embedded systems without adding complexities that aren't typically required (or, if they are required, they can be part of the application, not the monitor).

11.1.2 Ethernet MAC Address Storage

If your project has an Ethernet port, then it is going to need some type of MAC address storage. MicroMonitor supports a few different mechanisms for this…

- **monrc file:** The most convenient mechanism for storage of the MAC address is to simply put the "set ETHERADD …" command in the monrc file. This is certainly easy, and convenient; however, the MAC address is usually something that lives with the target, so the convenience of the monrc file may actually be a bad thing for storage of something like the MAC address because it is too easy to change.
- **etheradd:** Within each monitor image is a location in flash labeled "etheradd". This is space within the monitor binary flash area that is initialized to all 0xffs (erased). Typically it is at the base of the flash device and part of the reset.s source code. If the macro INCLUDE_STOREMAC is set to 1 in config.h, then when the monitor starts up it will look at that space and if it is erased, it will prompt the user to enter a MAC address that will then be burned into that space and used by the board at bootup. It is much less likely that the MAC address will be changed because it is not stored in a file; however, if the monitor binary is updated, this will be erased. Alternatively, the etheradd pointer could be initialized in the memory map linker file to some area of flash that is not part of TFS or part of the monitor binary space. This way both the monitor binary and TFS could be re-initialized without touching the stored MAC address.
- **hard coded in config.h:** If there is a value assigned to the DEFAULT_ETHERADD #define, then this will be used as the MAC address. This works fine, except that then each monitor binary has to be different at build time.
- **external serial device:** The MAC address is sometimes stored in an external serial EEPROM of some kind so that the rest of the system can be duplicated and all that has to be unique is the content of the EEPROM. This is supported with a call to the function *extGetEtherAdd()* in the common/monitor/ethernet.c file. If this functionality is not used, then the actual function (usually in the target-specific Ethernet driver file etherdev.c) should simply return 0; else it accesses the external storage device and returns a pointer to an ASCII string that represents the MAC address. This type of MAC storage is probably best for systems that are being mass-produced because the EEPROM devices can be purchased pre-programmed with some range of MAC addresses already burned in. Plus, this type of device may be set up to be read-only; hence, there is no way that the MAC address can be changed. This, by the way, can also serve as a serial number for the target.
- **boardinfo.c:** This source file can be used in the monitor build to store more than just the MAC address by adding entries to boardinfotbl[] (refer to comments in common/monitor/boardinfo.c). A variety of different settings can be stored in a dedicated flash sector, making them persistent even when MicroMonitor is updated or TFS is initialized. The disadvantage of this method is that a whole sector is wasted (however, if the flash device has a few small boot sectors, this may not be a problem).

These MAC address storage alternatives are mentioned in this porting section to make you aware of the fact that this should be considered during hardware design, because you may determine that your best solution is an external serial EEPROM. However, if it isn't considered all is not lost. MicroMonitor provides several alternatives for storage in the boot flash. When the monitor boots up, the function *getAddresses()* (in common/monitor/ethernet.c) looks for the MAC address in multiple places… First it looks to the environment as it would have been established by either the monrc file or the boardinfo initialization. If not found there, then it looks to some external device by calling *extGetEtherAdd()*. If this function returns NULL, it will look at what is pointed to by etheradd. Finally, if etheradd is all 0xffs, the monitor will use the value defined by DEFAULT_ETHERADD in config.h. If there is no definition there, the MAC address is set to 00:00:00:00:00:00 and the Ethernet device is not initialized.

11.1.3 Porting Simplicity Isn't Free

With the simplicity of the porting process comes some sacrifice in standalone monitor functionality. The two most significant points are...

- **The monitor does not use interrupts from the CPU**

From the porting point of view this is a real luxury. It makes it much easier to get a port to a new processor up and running. You don't have to worry about how a processor's interrupt handler mechanism is set up and you don't have to worry about making sure you save all the registers appropriately. Additionally, the serial port and Ethernet drivers are much less complex because everything is polled. Not having interrupts means that it is a bit more difficult (and somewhat inaccurate) to deal with elapsed time. If you have no clock reference, then you don't really know how much time has passed; however, to get around this a bit, there is a calibration that is made by the "sleep" command that allows you to adjust this timing a bit. Still, it is admittedly inaccurate. Keep in mind that the drivers in the monitor are not intended to be reused by the application anyway. If the overlaying application needs a real driver, then that application needs to provide it, and the monitor as a base development platform can provide a lot of help in doing that.

- **The monitor code assumes that initialized data is not writeable**

In many ports, the monitor code is running (fetching instructions) directly out of flash (see note1 below). The monitor does not copy any ".data" space to RAM, so it too is accessed in runtime from flash. This means that the monitor's initialized data is not writeable. Yep, this is something to be aware of but, once again, it makes the port much easier because now there is no need to figure out how the compiler toolset allows the data section to be remapped to RAM space and there is no need to copy the data section into RAM space (see note2 below). On the other hand, the monitor does assume that on startup the .bss section is initialized to zero (see start.c). This is ok because it is easy to clear a block of memory regardless of the toolset.

- **Note1:** the functions that perform write/erase operations on the flash are executed out of RAM. This is necessary because most flash devices do not allow the device to be modified if instructions are being fetched from it at the same time.
- **Note2:** initialized data in an embedded system must exist in non-volatile memory, and prior to application startup, if the data is to be writeable, it must be copied from non-volatile space to RAM. This is done so that at startup the data is properly initialized and at runtime the application has the option to change it. This means that some compiler-specific directives must be used to make sure that the data space in flash is copied from flash to some space in RAM where the application code will be able to access/modify it. This complexity is avoided in the monitor by carefully making sure that no initialized data is modified in the monitor code.

In some cases the monitor is specifically built so that it boots out of flash, but copies itself to RAM. If this is the case, then the above discussion on non-writeable initialized data is not applicable. Also, in this case, since the entire monitor runtime is in RAM space, there is no need to write flash drivers that copy themselves to RAM.

11.2 Getting Started

OK, we've covered the caveats, so if you're here that means you're going for it. Cool! Lets get started. First of all, regardless of the target you are building for, you need to build the tools that come with the uMon package first. For the most up-to-date information on this procedure, refer to the README file under umon_main/host (also refer to the top of Chapter 17 below). Next, if you're simply building a version of uMon from an already-existing port, then you can just run "*make rebuild*"[62] in the port-specific directory you plan to build. This will clobber any built files, generate a depends file, then build a 'boot' and 'ramtst' image that will be stored in the subdirectory "build_PLATFORM" (where PLATFORM is the name of the target).

If you're starting a port from scratch, then read on. If you're lucky, you'll find a port that is close to your target system, and you can start by copying that port directory to your new port directory. Otherwise, copy the template port directory to your new port directory and start from there. Even if your port is started from some other port that is "close" to yours, it is still a good idea to refer to the code in the template directory because it contains comments and recommendations around each of the needed functions.

To keep this text generic, the following steps reference the template port directory, which is simply a skeleton of source code into which the port-specific code can be added. Copy that entire directory to your new port directory. For the sake of this text, we'll refer to the port as 'sysXYZ', so in the case of the above subdirectory name, our name would be "build_sysXYZ".

Within any uMon port, there are 4 main sections of code that need to be modified:

[62] This is the equivalent of 'make clobber depend boot ramtst' (in that order).

- Code to handle CPU reset, basic exception handling and getting up to 'C' level. With this in place, you know you have a grasp on the whole build process. Ideally, your hardware will have some means of output (like an LED) that is easily accessible so that you can be sure your code is running. If you're lucky enough to have a JTAG (or similar) debugger, then that's even better!
- Code for a polled serial port driver. With the serial driver working, you immediately inherit the ability to download applications into RAM. One thing to make sure at this point is that your definition of LOOPS_PER_SECOND in config.h is close. This is important because many portions of the code use this count value for a delay loop; so if it isn't even close, facilities (like XMODEM) may not work efficiently.
- Code for a flash device driver. Once the flash device functions are installed, TFS will come up quickly.
- Code for a polled Ethernet driver. After this, TFTP, DHCP&BOOTP all just work! (really!).

Based on the ports already available, some of these sections may be re-usable from other ports that have already been made, so it is wise to browse through other ports prior to starting this work. If there is no port available that is close to what your target is, then start with the template port. This provides the basic set of empty files into which you can add the code discussed above.

11.3 Directory Structure

There are three top-level directories: umon_main, umon_ports and umon_apps, each of which has their own tree of sub-directories.

11.3.1 umon_main

This directory is the bulk of the uMon 1.0 code. All the target-independent code, plus the source for the host-based tools is under this directory. Immediately below umon_main is a README, and then the *target* and *host* directories. Refer to this README to get started.

11.3.2 umon_main/target

This directory contains the code that, in general, is applicable across all platforms. Below this space, are several sub-directories some of which are more target-independent than others…

- **common:** this directory contains 100% target-independent code. A subset of these files will be included in your target build, depending on the functionality you configure in.
- **cpu:** this directory contains a set of sub-directories each of which contains code that is specific to a CPU architecture. The directories include *arm*, *m68k*, *mips*, *ppc*, *sh* & *template*.
- **dev:** this directory contains device-specific files that may be re-useable across platforms.
- **flash:** this directory has 2 sub-directories. The original (boards) and newer (devices) flash drivers. A detailed discussion on these two directories is coming up in section 11.7.
- **make**: this directory contains files that are used by 'make' in the port-specific directory to provide common 'make' facilities.
- **misc:** this directory contains code that is typically not needed by the core monitor; however, it provides a few "extras" and/or "alternatives" to some of the code in the common directory. Some of this code is applicable to the monitor build while other code in this directory is useful for applications.
- **zlib:** this directory is very specific to the monitor's zlib-based decompression capability.

11.3.3 umon_main/host

In general, the tools that come with MicroMonitor can run on Win32, Solaris and Linux. This directory contains the code for each of the distributed tools (e.g. moncmd, newmon, ttftp, elf, etc…). Refer to the README at this level for details on how to build the tools for the PC, Linux or Solaris. Also refer to Chapter 17 for more information on building these tools. It's important to note that these tools must be built prior to building the port because the makefile assumes these tools are available at build time.

11.3.4 umon_ports

This directory contains sub-directories that, in general, are each applicable to a specific platform. At a minimum, this directory will contain a single target-specific port (the one you are working on or copying from), plus the template used to start a new port. For those who are involved in more than one port of MicroMonitor, additional target-specific directories will also reside here…

- **template:** this directory contains the set of stubs that must be filled in to complete a basic monitor port. Without modification it will build without error; however, the functionality within the stubs is NULL; hence it isn't ready for runtime.
- **sysXYZ:** this is the location of all code that is specific to the sysXYZ target.

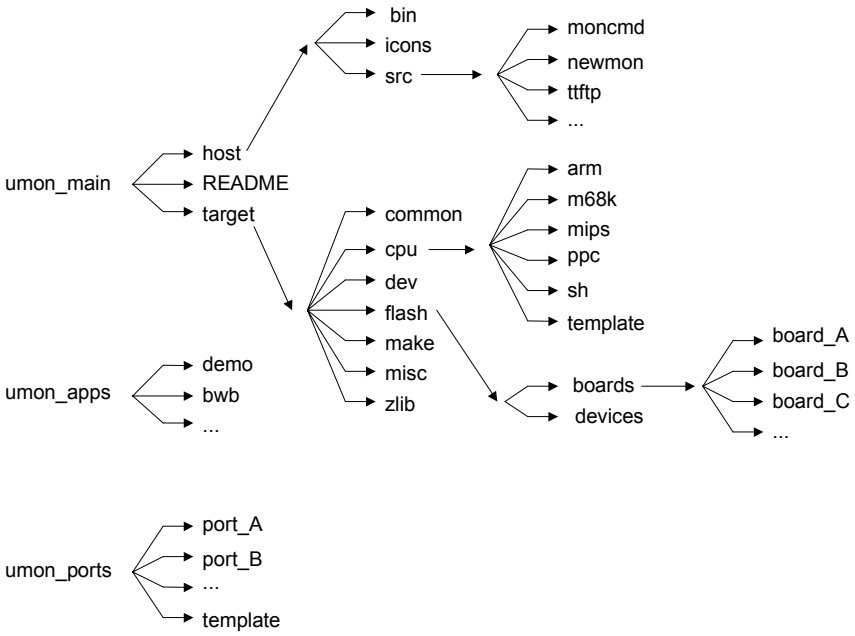
Note that the umon_ports directory is part of the official uMon source distribution. If your port is not destined to become one of the distributed ports, and you want to avoid having it conflict with the distribution, then it can be built outside of umon_ports directory and still link to the code under umon_main. Do this by simply making sure that the UMONTOP shell variable points to the umon_main directory path prior to invoking the port-specific make.

11.3.5 umon_apps/demo

This directory contains code that can be used to build a simple demonstration application for uMon-based targets running with MIPS, 68K, PowerPC and ARM architectures. The code provides a quick demonstration of the application's ability to hook up to uMon's file system, shell variables and command line. It also provides a quick demonstration of uMon's stack trace capability by providing a self-inflicted exception. It is not part of the porting process; however, will become handy when starting to build an application that runs on top of uMon. Refer to the README and makefile in that directory for build instructions.

11.3.6 The Source Tree

With the previous text in mind, Figure 6 is an overview of the monitor source tree for uMon 1.0. The umon_apps directory is not a requirement for the port itself; however, it provides a good starting point for building an application once the monitor port is completed. Note that the umon_ports directory is the directory that comes with the distributed uMon distribution.



• Figure 6: MicroMonitor Source Tree

11.4 The makefile

The makefile for any port is found under the port directory. We're assuming that we are working with a copy of the template port directory, named 'sysXYZ', so the makefile would be found under umon_ports/sysXYZ. The complete template makefile is shown in the following listings.

There are four main sections: environment setup, the source list, the build rules and miscellaneous. Notice that in each of the sections (not including miscellaneous), there is an “include” line that pulls in another file from umon_main/target/make. Each of the files included contain all of the generic stuff used by all make files; hence the actual target-specific makefile is relatively small.

```
001: ##############################################################################
002: #
003: # MicroMonitor Release 1.0 template board makefile.
004: #
005: # This file can be used as a starting point for a new port makefile.
006: # Structurally, this makefile should work for all ports, so the majority
007: # of the changes will be to modify the content of variables already defined.
008: # If you are transitioning to release 1.0 of MicroMonitor from an earlier
009: # verion, refer to the OLD_TO_NEW.txt file for a set of conversion steps.
010: #
011: # The VERY FIRST ERROR that this makefile is likely to generate will
012: # be due to the fact that the UMONTOP shell variable is either missing
013: # or not properly specified.  Make sure you start off by setting
014: # the UMONTOP variable to be set to the full path of the umon_main
015: # directory which contains all of the common source code for both target
016: # and host related builds for uMon.
017: #
018: # This template defaults to use a ppc-elf- GCC tool prefix.  Refer to
019: # CPUTYPE & FILETYPE variables below to change that.
020: #
021: ##############################################################################
022: #
023: # Build Variables:
024: # TOPDIR:
025: #   Set to the content of UMONTOP, which is an externally defined
026: #   shell variable assumed by this environment to be set to the full
027: #   path of the umon_main directory.
028: # PLATFORM:
029: #   ASCII name of the target platform (e.g. "Cogent CSB472")
030: # TGTDIR:
031: #   The name of the working directory that this port is to be built in.
032: # CPUTYPE/FILETYPE:
033: #   This combination of variables builds the GCC prefix (and is used for
034: #   a few other purposes.
035: #   Typical values for CPUTYPE are arm, ppc, m68k, mips and xscale.
036: #   Typical values for FILETYPE are elf, coff, rtems and aout.
037: # CUSTOM_FLAGS:
038: #   Establish custom portion of 'C' flags used for cross-compilation.
039: #   Refer to the file $(UMONTOP)/target/make/common.make for the set of
040: #   variables used (in addition to this one) to build the final CFLAGS
041: #   variable.
042: # CUSTOM_AFLAGS:
043: #   Similar to CUSTOM_FLAGS, this is used for assembler files.
044: # CUSTOM_INCLUDE:
045: #   Used to specify port-specific additions to the INCLUDES list.
046: #
047: PLATFORM        = TEMPLATE
048: TOPDIR          = $(UMONTOP)
049: TGTDIR          = template
050: CPUTYPE         = ppc
051: FILETYPE        = elf
052: CUSTOM_FLAGS    =
053: CUSTOM_AFLAGS   =
054: CUSTOM_INCLUDE  =
055:
056: ##############################################################################
057: #
058: # Memory map configuration:
059: # The following variables are used to establish the system's memory map.
```

```
060: # The values associated with these variables are substituted into
061: # the .ldt (.ld template) files to generate the .ld files actually used
062: # for the final linkage.  This allows the user to override these defaults
063: # without touching a memory map file.  Adjust them appropriately based on
064: # the target memory map.
065: #
066: # BOOTRAMBASE/BOOTRAMLEN:
067: # BOOTROMBASE/BOOTROMLEN:
068: #   Specify the base and length of RAM and ROM (i.e. flash) space used by
069: #   the version of the monitor that will reside (and run out of) boot flash.
070: # RAMTSTBASE/RAMTSTLEN:
071: #   Specify the base and length of RAM to be used by the "test" version of
072: #   the monitor that will reside entirely in RAM.
073: BOOTRAMBASE=0x3000
074: BOOTRAMLEN=0x7ffff
075: BOOTROMBASE=0xfff80000
076: BOOTROMLEN=0x7ffff
077: RAMTSTBASE=0x200000
078: RAMTSTLEN=0x7ffff
079:
080: include $(TOPDIR)/target/make/common.make
```

- **Listing 35: Makefile Environment Setup**

The environment setup is the most unique portion of the makefile (relative to other ports). This section includes the compiler flags, platform name, cpu and file type (used to create the toolset prefix[63]) etc… The CPUTYPE should be one of the supported MicroMonitor cpu subdirectories as listed in source tree section above. The second portion of the environment setup is the memory map variables. These variables are used in conjunction with a linker file template (*.ldt) to create the actual linker file (*.ld) for establishing the absolute address map of the monitor binary[64]. BOOTROMBASE and BOOTROMLEN are the address and length respectively of the flash space used by the monitor binary. BOOTRAMBASE and BOOTRAMLEN are the address and length respectively of the RAM space used by the monitor. RAMTSTBASE and RAMTSTLEN are the base and length of the version of the monitor built to run from ram for testing prior to burning into boot flash.

```
081:
082: ###########################################################################
083: #
084: # Build each variable from a list of individual filenames...
085: #
086: # LOCSSRC:
087: #   Local (in this directory) assembler files.
088: # LOCCSRC:
089: #   Local (in this directory) 'C' files.
090: #   Note regarding except_xxx.c and strace_xxx.c...
091: #   Prior to writing your processor-specific except_xxx.c and strace_xxx.c
092: #   check target's cpu directory to make sure it isn't already available.
093: #   If available, then change the filenames accordingly and move them to
094: #   the CPUCSRC filelist.  If you need to develop them, build them in
095: #   this port-specific directory, then upon completion they can be moved
096: #   to the cpu-specific directory so that they can be used by other ports.
097: # CPUSSRC:
098: #   CPU-specific assembler files in the main/target/cpu/CPU directory.
099: # CPUSSRC:
100: #   CPU-specific 'C' files in the main/target/cpu/CPU directory.
101: # COMCSRC:
102: #   Common 'C' files found in the main/target/common directory.
103: # IODEVSRC:
104: #   Device-specific files found in main/target/dev directory.
105: # FLASHSRC:
106: #   The flash driver file found in main/target/flash/devices directory.
```

[63] If the toolset you're using doesn't follow the "cpu-filetype" convention (used by Microcross), then specify TOOL_PREFIX in this section to force the prefix.

[64] Refer to the 'vsub' command description in the list of host based commands.

```
107: #
108: LOCSSRC     = reset.S
109: CPUSSRC     =
110: LOCCSRC     = cpuio.c etherdev.c except_template.c strace_template.c
111: COMCSRC     = arp.c bbc.c cast.c cache.c chario.c cmdtbl.c crypt.c \
112:               docmd.c dhcp_00.c dhcpboot.c edit.c ee.c env.c ethernet.c \
113:               flash.c genlib.c icmp.c if.c ledit_vt100.c monprof.c \
114:               mprintf.c memcmds.c malloc.c moncom.c memtrace.c misccmds.c \
115:               misc.c password.c redirect.c reg_cache.c sbrk.c start.c \
116:               symtbl.c tcpstuff.c tfs.c tfsapi.c tfsclean1.c tfscli.c \
117:               tfsloader.c tfslog.c tftp.c xmodem.c gdb.c
118: CPUCSRC     =
119: IODEVSRC    =
120: FLASHSRC    = am29lv160d_16x1.c
121:
122: include $(TOPDIR)/target/make/objects.make
123:
124: OBJS    =   $(LOCSOBJ) $(CPUSOBJ) $(LOCCOBJ) $(CPUCOBJ) $(COMCOBJ) \
125:             $(FLASHOBJ) $(IODEVOBJ)
126:
```

- **Listing 36: Makefile Source List**

The source list is made up of 7 variables:

LOCSSRC	list of assembler (.S) files in the port directory
LOCCSRC	list of 'C' files in the port directory
COMCSRC	list of 'C' files in the common (umon/umon_main/target/common) directory
CPUSSRC	list of assembler files in the cpu-specific directory (in this case umon/umon_main/target/cpu/m68k)
CPUCSRC	list of 'C' files in the cpu-specific directory
IODEVSRC	list of 'C' files that are reusable device interfaces under umon/umon_main/target/dev)
FLASHSRC	list of 'C' files that make up the flash device driver

Usually from one port to the next, the COMCSRC list is the same. The other variables allow you to use files in common space as well as files in the port's specific directory. To support both the new and old flash device driver source code, the shell variable FLASHDIR can be set to point to one of the directories under umon/umon_main/flash/boards; otherwise, it will be assumed that the FLASHSRC files reside under umon_main/flash/devices (i.e. the new model, discussed below).

```
127: ###########################################################################
128: #
129: # Targets...
130:
131: # boot:
132: # The default target is "boot", a shortcut to $(BUILDDIR)/boot.$(FILETYPE).
133: # This builds the bootflash image that can be used by 'newmon' to
134: # load a new version onto an already running system.
135: #
136: boot:   $(BUILDDIR)/boot.$(FILETYPE)
137:     @echo Boot version of uMon built under $(BUILDDIR) ...
138:     @ls $(BUILDDIR)/boot*
139:
140: # ramtst:
141: # A shortcut to $(BUILDDIR)/ramtst.$(FILETYPE).  This is a version of uMon
142: # that resides strictly in RAM and is used for two main purposes:
143: # 1. To test new monitor features prior to burning the boot flash.
144: # 2. To be downloaded into the RAM space of a board that has no programmed
145: #    boot flash.  This provides a running monitor that can then accept
146: #    an incoming bootflash image using 'newmon'.
147: #
```

```
148: ramtst: $(BUILDDIR)/ramtst.$(FILETYPE)
149:     @echo Ram-resident test version of uMon built under $(BUILDDIR) ...
150:     @ls $(BUILDDIR)/ramtst*
151:
152: $(BUILDDIR)/boot.$(FILETYPE): $(BUILDDIR) $(OBJS) libz.a makefile
153:     $(MAKE_MONBUILT)
154:     $(MAKE_LDFILE) \
155:         BOOTRAMBASE=$(BOOTRAMBASE) BOOTRAMLEN=$(BOOTRAMLEN) \
156:         BOOTROMBASE=$(BOOTROMBASE) BOOTROMLEN=$(BOOTROMLEN)
157:     $(LINK) -e coldstart $(OBJS) monbuilt.o libz.a $(LIBGCC)
158:     $(MAKE_BINARY)
159:     $(MAKE_GNUSYMS)
160:
161: $(BUILDDIR)/ramtst.$(FILETYPE): $(BUILDDIR) $(OBJS) libz.a makefile
162:     $(MAKE_MONBUILT)
163:     $(MAKE_LDFILE) \
164:         RAMTSTBASE=$(RAMTSTBASE) RAMTSTLEN=$(RAMTSTLEN)
165:     $(LINK) -e coldstart $(OBJS) monbuilt.o libz.a $(LIBGCC)
166:     $(MAKE_BINARY)
167:     $(MAKE_GNUSYMS)
168:
169: include $(TOPDIR)/target/make/rules.make
170:
```

- **Listing 37: Makefile Build Rules**

There are two main rules: boot and ramtst. The boot rule builds the image that is destined for the boot flash. The ramtst rule builds the image that can reside in RAM and is useful for testing new features built into uMon prior to actually installing to the boot flash. Notice that both boot and ramtst point to other targets and are used primarily for simplicity on the make command line. Each of the main rules are pretty standard for all ports. The majority of the variables (i.e. BUILDDIR, MAKE_MONBUILT, MAKE_LDFILE, etc…) are defined in $(TOPDIR)/target/make/common.make. Of particular interest is the MAKE_LDFILE variable. This line (154-157 for boot & 163-164 for ramtst) creates a .ld file from the specified variables and the template .ldt file using a simple tool called 'vsub' that comes with the monitor package[65].

```
171: ##########################################################################
172: #
173: # Miscellaneous...
174: #
175: ######
176: #
177: # cscope_local:
178: # Put additional files here that should be included in the cscope
179: # files list.  This is called before the generic cscope file builder,
180: # so it should create the cscope.files file.
181: #
182: cscope_local:
183:     >cscope.files
184:
185: ######
186: #
187: # help_local:
188: # Add text here as needed by the port.
189: #
190: help_local:
191:     @echo "This template defaults to using ppc-elf as the tool prefix."
192:     @echo "To override this default modify CPU & FILETYPE variables."
193:     @echo
```

[65] The function of vsub could probably be done with some combination of sed/awk commands; however I find it easier to just write small C programs to accomplish this kind of stuff.

- **Listing 38 : Makefile Miscellaneous**

The minimum 'miscellaneous' section includes the cscope_local and help_local rules. Cscope (available at http://sourceforge.net/projects/cscope) for those who haven't used it, is a tool that allows you to browse through large source trees. Cscope looks for a file called cscope.files which is simply a listing of all files that cscope should consider to be part of the source tree. The rule "make cscope" will build a file for all of the files in the source list section; however, if for some reason some of the files are not in that section, they can be listed here for inclusion in the list. The help_local target is provided to allow the writer of a port to include port-specific notes that will be output when the rule "make help" is invoked at the command line. Also, if additional rules are added to the makefile, then they can be elaborated on here for use by others.

11.4.1 The Make File's 'help' Rule

There are a lot of pre-configured rules that make much of the MicroMonitor maintenance simpler. To keep track of these rules, the 'help' rule will dump a summary of each of the available rules. The output shown in Figure 7 is an example...

```
make help
The following generic make-targets are available:
boot:     Build a bootrom-resident version of uMon
ramtst:   Build a ram-resident version of uMon for testing
clobber:  Remove all files built by this makefile.
clean:    Remove all object files built by this makefile.
ctags:    Build a tags file for use by most source editors.
cscope:   Build a cscope.files file for use by cscope.
libgcc:   Dump the libgcc used.
depend:   Create a dependency list (file=depends) for the build.
emap:     Dump a map of the build using the 'elf' tool.
map:      Dump a map of the build using objdump.
          This requires BUILD=xxxx to be specified on the
          make command line.  The value of xxxx will usually
          be 'boot', 'ram' or 'ramtst'.
          Example: make BUILD=boot map
monsym:   Create the symbol table file (*.usym) used by
          MicroMonitor. See note in 'map' above regarding BUILD.
dis:      Create a source/disassembly file (*.dis) of the image.
          See note in 'map' above regarding BUILD.
rebuild:  Concatenation of "clobber, depend, boot and ramtst"
newmon:   Run through the steps needed to burn a new monitor
dld:      Download the ramtst image to the target.
rundisable:
          Rebuild with the tfs run functionality disabled.
bdibuild: Build the files used for BDI2000 disaster recovery.

Build-specific outputs for sysXYZ are suffixed as follows:
  *.bin:    Raw binary image of build, suitable for transfer
            directly to memory.
  *.elf:    ELF-formatted image.
  *.dis:    Disassembly of uMon build.
  *.gsym:   Symbol table formatted by gnu tools (nm).
  *.usym:   Symbol table formatted by uMon tools (monsym).
and will be placed in the directory "build_sysXYZ".
```

- **Figure 7: Output of 'make help'**

11.5 The config.h file:

At this point we are ready to start modifying code. The config.h file is a file that is included by every source file in the monitor. It contains (among other things) the defines that establish the features in the monitor that are to be enabled. After setting the PLATFORM_NAME and CPU_NAME, the next thing to do is to reduce the complexity of the monitor by turning off the majority of the features established by the sysXYZ/config.h file[66].

[66] Obviously the port is already complete, so there's no need to modify the source code. Refer to the files as needed.

In this file there is a block of INCLUDE_XXX macros (where XXX is some feature-specific string). Following is what can be used as the basic configuration[67]...

```
#define FORCE_BSS_INIT          1

#define INCLUDE_MALLOC          1
#define INCLUDE_MEMCMDS         1
#define INCLUDE_SHELLVARS       1
#define INCLUDE_XMODEM          1
#define INCLUDE_EDIT            0
#define INCLUDE_DISASSEMBLER    0
#define INCLUDE_UNZIP           0
#define INCLUDE_ETHERNET        0
#define INCLUDE_ICMP            0
#define INCLUDE_TFTP            0
#define INCLUDE_TFS             0
#define INCLUDE_FLASH           0
#define INCLUDE_LINEEDIT        0
#define INCLUDE_DHCPBOOT        0
#define INCLUDE_TFSAPI          0
#define INCLUDE_TFSSYMTBL       0
#define INCLUDE_TFSSCRIPT       0
#define INCLUDE_TFSCLI          0
#define INCLUDE_EE              0
#define INCLUDE_GDB             0
#define INCLUDE_STRACE          0
#define INCLUDE_CAST            0
#define INCLUDE_REDIRECT        0
#define INCLUDE_QUICKMEMCPY     0
#define INCLUDE_PROFILER        0
#define INCLUDE_BBC             0
#define INCLUDE_MEMTRACE        0
#define INCLUDE_STOREMAC        0
#define INCLUDE_USRLVL          0
```

• Listing 39 : The INCLUDE_XXX Definitions

This configuration sets up a very basic monitor... a serial port driver with Xmodem for download and memory commands for interfacing to target memory. Refer to the file common/monitor/inc_check.h for the most up-to-date list of these definitions and their meaning. The inc_check.h file is updated each time some new feature is added to (or removed from) the monitor. This file attempts to enforce the inclusion (or exclusion) of the INCLUDE_XXX macros regardless of whether or not the feature is to be used. The point is to just make sure the builder of the monitor is aware of the feature. With the core set of ports I attempt to keep each config.h file in sync with the inc_check.h header; however, it is very possible that at build time, there will be an error like…

```
#error "INCLUDE_XXX must be defined in config.h"
```

(where XXX is some feature that the inc_check.h file is "reminding" the user of). To eliminate this error, simply add that INCLUDE_XXX line to your config.h. This forces you to be aware of the feature, but still allows you to simply disable it by setting the definition to zero.

Referring to Listing 39, also note the definition of FORCE_BSS_INIT. This is used by umon_main/target/common/start.c to unconditionally clear the monitor's BSS space at reset. Later, when things settle down a bit, this definition can be removed so that a soft reset of the monitor does not wipe out the existing environment.

[67] The smallest practical footprint would have INCLUDE_XMODEM or INCLUDE_ETHERNET&INCLUDE_TFTP (with all other macros set to zero).

If this is the first attempt of a port to this processor, then spend some time looking for some example code from the CPU manufacturer's website and other open source projects. This is usually not too hard to find. For the very first build attempt, don't worry about the exception handlers. Once the basic bootup (memory configuration and serial port) is stable, then add the exception handling stuff. For the CPUs currently supported by uMon, the code would be found under umon_main/target/cpu/XXX (where XXX is your CPU architecture) and the files are named something like "except_zzz.c" and vectors_zzz.S.

11.6 Runtime Execution, FLASH or RAM?

The monitor can be built to run out of boot flash at runtime, or it can be built to copy itself from boot flash to RAM and run from RAM at runtime. Refer to different targets to see the different implementations. In general, the decision is based on the needs of the system. In some cases, RAM is limited, so space cannot be allocated to the monitor, hence it is left in flash. In some cases the boot flash is a slow 8-bit wide device, so fetching from that device at runtime is inefficient, so the monitor is copied to RAM for speed. The configuration used depends on the target it is running on.

If running from flash, then the monitor is built as a single image (this is the case for the template). If running from RAM, then the monitor is built as two images. The first image is the code that must start off in boot flash. The second image is converted to an array that is included in the first image's source file cstart.c (more on that later). The first image then simply copies that array to some pre-defined location in RAM and branches to that point[68]. There are several reasons why one solution may be more appropriate than the other, it just depends on what you're building and what constraints you have to deal with, so just pick the one that works best for your system.

11.6.1 Run From Flash

This mode is the easier of the two to boot up, simply because there's no image relocation or anything fancy to do; hence, less chance of error[69]. Even if you plan to use the RAM-based monitor, depending on your confidence level with the target and CPU, it may make sense to start with this mode just to get over the very basic startup hump of the port. The build strategy is straightforward. A single image is generated and configured to run the reset location in the boot flash. Care must be taken to make sure that the entrypoint of the image is at the location in the memory map that the CPU will fetch from when it is reset. The startup file[70] for this mode is almost always called "reset.s" or "rom_reset.s".

The startup point of the reset code for ROM-resident monitors is a coldstart and a warmstart tag, both of which must be in assembler. The coldstart tag is the first significant piece of code that should be pointed to by the reset vector of the CPU[71]. The primary difference between coldstart and warmstart is that warmstart must be callable by the firmware to do a soft reset of the target; hence, warmstart must do as much re-initialization of the core CPU as possible so that a call to warmstart from the firmware will be similar a hard reset. The warmstart tag assumes it is being passed an argument that must be stored in a register and used later by the 'C' function start() to determine how to initialize the rest of the system. The coldstart code must enter warmstart the same way a C function would call warmstart with a single parameter. That parameter indicates that the target is in the INITIALIZE state. As a result, a hard reset (or power up) will invoke coldstart and coldstart will eventually call warmstart with INITIALIZE as the parameter. A firmware-generated reset will invoke warmstart with various other parameter options (including INITIALIZE) depending on the source of the call…

```
coldstart:
        ...
        * Load INITIALIZE into the appropriate location (or register) such that
          warmstart thinks it is being called with the parameter "INITIALIZE",
          i.e. warmstart(INITIALIZE).
        * Jump to warmstart
        ...

warmstart:
```

[68] Note that eventually, this dual-image mode will also support the ability to decompress the array out of flash and into RAM for execution (not there yet!).

[69] The only caveat of this mode is that usually the flash drivers must reside in RAM, so they will have to be relocated (more on that later).

[70] In this context, the "startup file" simply refers to the file that contains the code that is fetched by the CPU at reset.

[71] In some ports the "coldstart" tag may be replaced with "reset".

```
        * Move parameter to some location from which it can be retrieved prior
          to calling start().
        ...
        * Initialize as much of the CPU as possible so that a call to warmstart
          looks like a real reset.
        * Initialize RAM space for use by the stack.
        ...

gotoC:
        * Set the stack pointer to point to valid RAM space.
        * Retrieve the parameter stored away by warmstart and place that value into
          the location (register or stack offset) such that the start()
          function sees it as its parameter.
        * Jump to start()
```

• Listing 40 : Coldstart/Warmstart PseudoCode

The file "reset.S" under the umon_ports/template directory contains a good, target-independent starting point for a new port.

Storage of moncomptr and MAC Address

The startup code contains the moncomptr and the optional flash-resident storage location for the MAC address of the target…

```
01: .global etheraddr
02: .global moncomptr
03: .extern moncom
04:
05:
06: /* moncomptr:
07:  * Pointer to the moncom function, used to link application to monitor.
08:  */
09: moncomptr:
10:     .long   moncom
11:
12: /* etheraddr:
13:  * Location that could be used to store a fixed MAC
14:  * address.
15:  */
16: etheraddr:
17:     .byte 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff
18:     .byte 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff
19:     .byte 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff
```

• Listing 41: Snippet from reset.s

Referring to Listing 41, lines 9&10 provide a stable location for the "well known address" used when the application calls monConnect() (first discussed in section 8.1). The startup code should be at a position in the memory map such that a re-build of the monitor will not cause the address of moncomptr to change. The value of moncom (the name of the function that moncomptr points to) can change; however, the location of moncomptr should be stable. Lines 16-19 provide an alternate location for the MAC address to be stored[72]. The most convenient place for the MAC address to be initialized is in the monrc file; however, the default monitor code supports several alternatives…

```
01: if (!(Etheradd = getenv("ETHERADD"))) {
02:    if (!(Etheradd = extGetEtherAdd())) {
03:           if (etheraddr[0] != 0xff)
04:                  Etheradd = etheraddr;
05:           else
```

[72] Depending on the port, the code of lines 6-10 may be "include moncomptr.S" and the code of lines 12-19 may be "include etheraddr.S". These two .S files are the only two .S files under umon_main/target/common because they can be used by all ports.

```
06:                   Etheradd = DEFAULT_ETHERADD;
07:     }
08:     setenv("ETHERADD", Etheradd);
09: }
```

• Listing 42 : Snippet of getAddresses() Function

The code of Listing 42 is a snippet taken from the monitor function getAddresses() in the Ethernet startup code of the monitor (umon_main/target/common/ethernet.c). This portion of the function is the algorithm used to set the "Etheradd" pointer in the monitor to some pre-configured MAC address string. The monitor makes 3 different attempts to find a MAC address in the target system, then if all three fail, the default from config.h is used…

- **First attempt:** (line #1) check the environment variable ETHERADD, which would have been set by the monrc file.
- **Second attempt:** (line #2) call the extGetEtherAdd() function, it would return a pointer to the ASCII-based MAC address if it was active. This allows the port-specific code to include a port-specific mechanism (e.g. serial EEPROM) for retrieval of the board's MAC address.
- **Third attempt:** (line #3) if the first location of the etheraddr array in reset.s is not 0xff, then assume that array contains the ASCII-based MAC address string.
- **If all else fails:** (line #6) use DEFAULT_ETHERADD specified in config.h. Note that if DEFAULT_ETHERADD is not set in config.h, then it defaults to 00:00:00:00:00:00 (see umon_main/target/common/ether.h).

11.6.2 Run From RAM

This mode is slightly more complicated to get started because of the fact that the monitor now has to boot up from flash and copy an image to RAM, then jump into the entrypoint of that image. The build strategy for this mode is to generate 2 images:

- an image of a full monitor executable that would reside in RAM
- an image that can do the very basic bootup of the CPU (core cpu and memory initialization), copy the RAM-destined image to its pre-configured location, and then jump into the entrypoint of that image.

The RAM-destined image is the one that will contain the complete monitor; while the flash-resident image will be minimal. Its job ends as soon as it copies the RAM-based image to RAM and transfers control to it.

The makefile

Referring to Listing 43, the top level target (line 10) of the makefile directs make to build the RAM-based image first. As the names imply, one target builds the RAM-based image and one target builds the ROM-based[73] boot image. The RAM-based image must be built first so that the result of that build can be included in the ROM-based image's build. Lines 38-40 convert the RAM-based image to a file called umon.c. The "bin2array" tool (one of the tools that comes with the monitor) simply converts the RAM binary image into a C-array (in this case, umon.c). This file is included in the file umon_main/target/common/cstart.c (see Listing 44). The file cstart.c is then included in the ROM based build, and called by the ROM-based startup code to do the copy to RAM and then branch to the entrypoint. The "UMON_RAMBASE $(RAMBASE)" definition at line 38 is coordinated with the RAM image's memory map linker file and is what is used by the cstart code as the base address into which the array is copied. The "UMON_START $(RAMBASE)" definition establishes the entrypoint of the RAM based image and is used, once again, by cstart() to branch to the starting point of the RAM-based image. Notice that in this case, the base and start are the same address.

```
05: # boot:
06: # The default target is "boot", a shortcut to $(BUILDDIR)/boot.$(FILETYPE).
```

[73] Note that the term "ROM" is used here only to distinguish a difference between it and the RAM based image. In most real systems, flash is the underlying memory type.

```
07: # This builds the bootflash image that can be used by 'newmon' to
08: # load a new version onto an already running system.
09: #
10: boot:   ram $(BUILDDIR)/boot.$(FILETYPE)
11:     @echo Boot version of uMon built under $(BUILDDIR) ...
12:     @ls $(BUILDDIR)/boot*
13:
14: ram:    $(BUILDDIR)/ram.$(FILETYPE)
15:     @echo Ram version of uMon built under $(BUILDDIR) ...
16:     @ls $(BUILDDIR)/ram\.*
17:
18: $(BUILDDIR)/boot.$(FILETYPE): $(BUILDDIR) $(BOOTOBJS) libz.a makefile
19:     $(CC) $(COMMON_AFLAGS) $(CUSTOM_AFLAGS) \
20:         $(COMMON_INCLUDE) -orom_reset.o rom_reset.S
21:     $(MAKE_LDFILE) \
22:         BOOTRAMBASE=$(BOOTRAMBASE) BOOTRAMLEN=$(BOOTRAMLEN) \
23:         BOOTROMBASE=$(BOOTROMBASE) BOOTROMLEN=$(BOOTROMLEN)
24:     $(LINK) -e reset --oformat elf32-bigmips --no-warn-mismatch $(BOOTOBJS)
25:     $(MAKE_BINARY)
26:     $(MAKE_GNUSYMS)
27:     $(MAKE_MONSYMS)
28:
29: $(BUILDDIR)/ram.$(FILETYPE): $(BUILDDIR) $(RAMOBJS) libz.a makefile
30:     $(MAKE_MONBUILT)
31:     $(MAKE_LDFILE) \
32:         RAMBASE=$(RAMBASE) RAMLEN=$(RAMLEN) \
33:         MACADDRBASE=$(MACADDRBASE) IPADDRBASE=$(IPADDRBASE)
34:     $(LINK) -e coldstart --oformat elf32-bigmips --no-warn-mismatch \
35:         $(RAMOBJS) monbuilt.o libz.a $(LIBGCC)
36:     $(TOOLBIN)/$(FILETYPE) -m $@
37:     $(TOOLBIN)/$(FILETYPE) -B $(BUILDDIR)/ram.bin $@
38:     $(TOOLBIN)/bin2array -D "UMON_RAMBASE $(RAMBASE)" \
39:         -D "UMON_START $(RAMBASE)" -o $(BUILDDIR)/umon.c \
40:         -a umon $(BUILDDIR)/ram.bin
41:     rm -f cstart.o
42:     $(MAKE_GNUSYMS)
43:     $(MAKE_MONSYMS)
```

• Listing 43 : Snippet of RAM-Based Monitor Makefile

Note that this ram-based build has the same “ramtst” rule as does the standard build. It would be mapped to a second section of RAM that is not being used by the running ram-based image.

The cstart() Function

The cstart() function is likely to be the only ‘C’ function in the ROM-based portion of a “run-from-ram” build of uMon. After the ROM-based boot code does basic initialization (which must include initialization of the RAM space into which the image will be copied), the cstart() function is called. Referring to Listing 44, notice that the cstart() function simply includes the umon.c file. This file contains the array and the defines needed by cstart() to copy and jump into the RAM-based image. This mechanism of image copying is convenient because it is independent of the toolset and CPU. The intent is that future versions of MicroMonitor will support decompression of this array into RAM; hence, reducing the size of the flash space needed for storage of the MicroMonitor image.

```
36: /* The umon.c file is simply an array that contains the monitor
37:  * image and the necessary information needed to do the copy.
38:  */
39: #include "umon.c"

...

103: void
104: cstart(void)
105: {
```

```
106:     register char *cp1, *cp2, *end2;
107:     void (*entry)();
108:
109:     entry = (void(*)())UMON_START;
110:
111:     /* Copy image from boot flash to RAM, then verify the copy.
112:      * If it worked, then jump into that space; else reset and start
113:      * over (not much else can be done!).
114:      */
115:     memcpy((char *)UMON_RAMBASE,(char *)umon,(int)sizeof(umon));
116:
117:     /* Verify the copy...
118:      */
119:     cp1 = (char *)UMON_RAMBASE;
120:     cp2 = (char *)umon;
121:     end2 = cp2 + (int)sizeof(umon);
122:     while(cp2 < end2) {
123:         if (*cp1 != *cp2) {
124: #ifdef CSTART_ERROR_FUNCTION
125:             extern void CSTART_ERROR_FUNCTION();
126:
127:             CSTART_ERROR_FUNCTION(cp1,*cp1,*cp2);
128: #endif
129:             entry = RESETFUNC();
130:             break;
131:         }
132:         cp1++; cp2++;
133:     }
134:     entry();
135: }
```

• **Listing 44: Snippet of the File cstart.c**

Multiple Entrypoints

Since there are two images, there are essentially two different startup points, one for the ROM-resident image and one for the RAM-resident image. To organize this, the names of the startup files change from simply being "reset.s" to rom_reset.S (for the ROM-based image) and ram_reset.S (for the RAM-based image), then the file reset.S can be used to contain common assembly code used by both ram_reset.S and rom_reset.S. The purpose behind coldstart and warmstart (discussed in section 11.6.1) is the same; however, now the coldstart code in ram_reset.S is not the address of the CPUs reset vector. It is the entrypoint used by the ROM-resident startup code to jump into the RAM-resident monitor. The ram_reset.S file will contain the moncomptr value because this is a pointer to the monitor library accessible to the application, and the RAM-based monitor image is the one that will be active when the application is running. The rom_reset.S code will contain the ROM-resident, storage location for the MAC address. It is put here simply because this code is resident to the flash, and the value is stored in flash. Its location must be included in the RAM-based image's memory map linker file. Note that this implies that the code of Listing 41 will be broken up into two different files…

```
.global etheraddr

reset_vector:
       * Initialize CPU core
       * Initialize RAM
gotoC:
       * Establish stack pointer
       * Call cstart()

/* etheraddr:
 * Location that could be used to store a fixed MAC
 * address.
 */
etheraddr:
    .byte 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff
```

```
    .byte 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff
    .byte 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff, 0xff
```

• **Listing 45: Pseudocode for rom_reset.s**

```
.global moncomptr
.extern moncom

/* moncomptr:
 * Pointer to the moncom function, used to link application to monitor.
 */
moncomptr:
    .long   moncom

coldstart:
       See Listing 40 for details.
```

• **Listing 46: Pseudocode ram_reset.s**

11.6.3 In General…

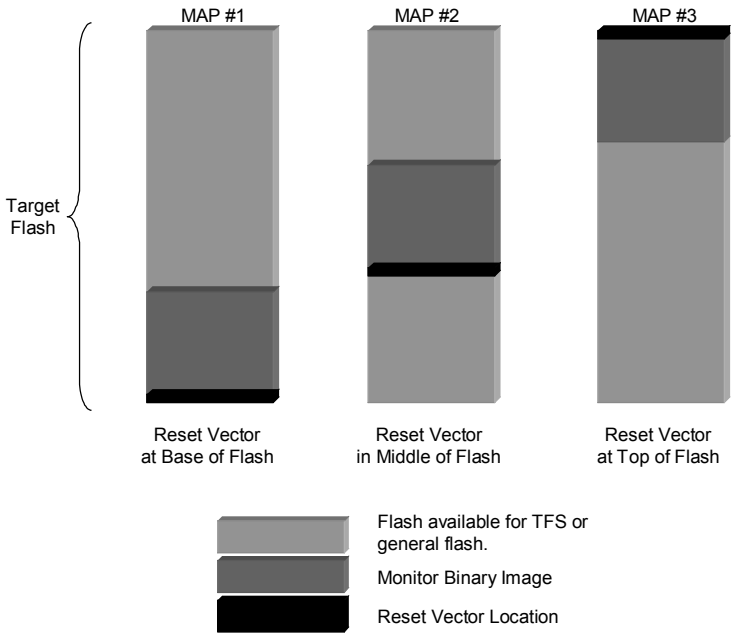
Regardless of the startup mode used, typical issues to deal with at bootup are disabling/clearing cache, disabling MMU (if any), disabling interrupts, adjusting boot flash access timing (chip selects) and configuring the on-board RAM memory area (typically a DRAM controller). Once this startup code is established and organized, the remaining porting issues are independent of which startup mode is used (except for the need to copy flash drivers to RAM).

11.7 The Memory Map

There are a lot of different ways the monitor can be configured to use flash and ram in the target system. The text of section 11.6 above, discusses one aspect of the monitor's memory map to establish whether or not the instructions of the monitor itself are fetched from flash or ram. As that discussion implies, it is very dependent on the needs and configuration of the target. This discussion is also very dependent on the target's memory configuration. The point to note at the start is that MicroMonitor is very flexible with regard to how much and where memory is used in the target.

11.7.1 The Boot Sector

This is the only portion of the memory map that is fixed. This is not a requirement of MicroMonitor as much as it is a fundamental requirement of the CPU that MicroMonitor is being run on. All microprocessors have some defined entrypoint or reset vector address that establishes the location in memory where the microprocessor will fetch its initial instruction immediately after a reset or power up. Since MicroMonitor is the code that executes immediately after a reset, its entrypoint must be coordinated with the microprocessor's reset vector.

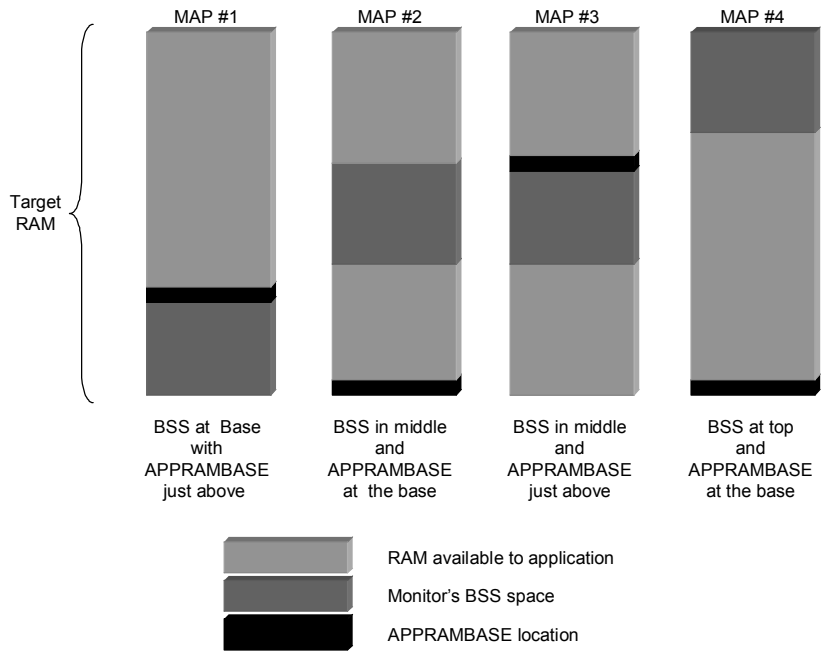


• **Figure 8: Reset Vector Location & MicroMonitor Entrypoint**

Referring to Figure 8, this reset vector address can be anywhere in the memory map. On some processors it's at the bottom of memory, some at the top and some are somewhere in the middle. The point to be aware of here is that MicroMonitor will not necessarily be starting at the base of a flash device. It may be at the top (map #3), bottom (map #1) or somewhere in the middle (map #2). If the reset vector address for the CPU is at the base or middle memory, then the entrypoint of the monitor is the lowest address in the monitor's flash-resident text space. If the reset vector is at the top of memory, then it will typically be a branch or jump instruction that brings the instruction pointer down a bit lower to the entrypoint of the monitor code in reset.s.

11.7.2 Location of MicroMonitor's BSS Space

In simple systems, the monitor's BSS is at the base of RAM. The APPRAMBASE shell variable is established to contain the base address of the memory that is available for use by the application (just above the end of the monitor's BSS space). This is shown pictorially in map #1 of Figure 9. In this case, all of the space above APPRAMBASE is useable by the application. The stack and heap used by MicroMonitor are included within the monitor's own BSS space (allocated as pre-configured arrays of memory); hence, there is no need to deal with multiple non-contiguous blocks of RAM in use by the monitor. This keeps it simple and easy to move around.

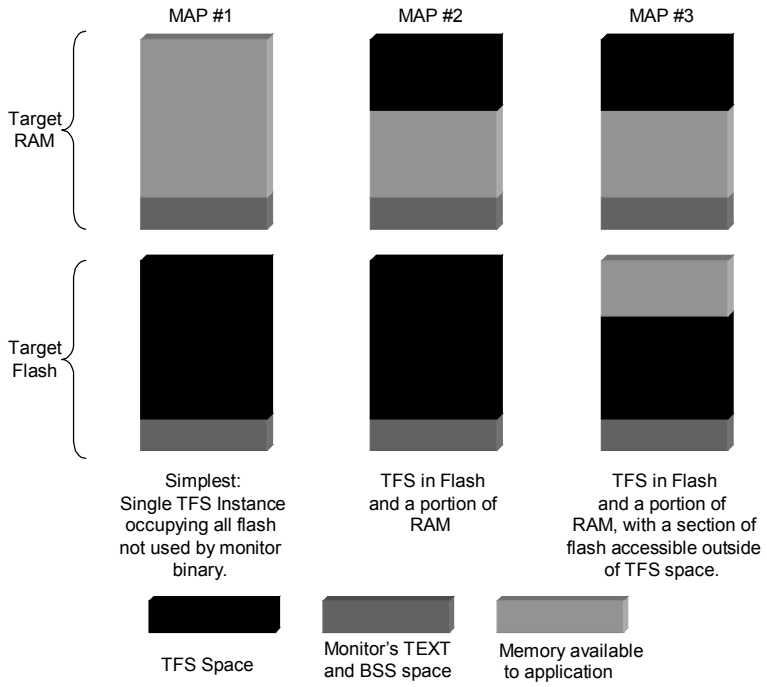

MAP #1
MAP #2
MAP #3
MAP #4
Target RAM
BSS at Base with APPRAMBASE just above
BSS in middle and APPRAMBASE at the base
BSS in middle and APPRAMBASE just above
BSS at top and APPRAMBASE at the base
RAM available to application
Monitor's BSS space
APPRAMBASE location

• **Figure 9: Different BSS/APPRAMBASE Configurations**

In some cases, the application that is being run by MicroMonitor is required to reside at the base of the RAM space on the target. It then may seem logical to put MicroMonitor's BSS up at the top of the target's RAM space and have the APPRAMBASE shell variable point to the base of RAM (shown in map #4 of Figure 9). In some cases this is appropriate; however in others, that same application may be using the top of memory for its stack and heap and as a result, MicroMonitor's BSS space must be located somewhere in the middle (maps 2 & 3 of Figure 9). The point of all this is that it really doesn't matter. By default, the APPRAMBASE value is automatically configured (refer to the function init0 in umon_main/target/common/start.c) to point to an address just above the end of MicroMonitor's BSS space; however, the APPRAMBASE value can be forced to point to wherever is appropriate by using the APPRAMBASE_OVERRIDE macro in config.h (discussed in Section 11.10.4 below).

11.7.3 Configuration and Location of TFS

TFS can be configured to run in multiple sections of memory that are not necessarily contiguous nor are they required to be flash. Details on how to configure TFS are in Section 11.10 below; however, since we're discussing memory maps here, it seems logical to at least mention the various configurations that TFS supports. TFS is simply a format applied to a block of flash (or RAM). Several instances (or "devices", as they are called in Section 11.10) of TFS can be configured within the memory space of the target. The simplest and most common example is to have a portion of the flash allocated to the monitor binary and the remainder of that flash allocated to TFS (refer to map #1 below).


MAP #1
MAP #2
MAP #3
Target RAM
Target Flash
Simplest: Single TFS Instance occupying all flash not used by monitor binary.
TFS in Flash and a portion of RAM
TFS in Flash and a portion of RAM, with a section of flash accessible outside of TFS space.
TFS Space
Monitor's TEXT and BSS space
Memory available to application

Depending on the system needs, multiple TFS devices can reside in flash, NVRAM and RAM. Plus, some space in flash can be left as free flash space for use by the application however it sees fit (refer to the monitor API functions mon_flasherase(), mon_flashwrite() and mon_flashinfo() in Chapter 16). The only requirement is that each instance of TFS must have its own contiguous block of memory that has allocated the appropriate amount of space for the spare sector. Even the NVRAM based TFS uses a spare sector because it will defragment the RAM as if it was flash. Once again, the point is that there is a lot of flexibility when configuring the flash space for your MicroMonitor based target.

11.7.4 Memory Map Summary

The point of this discussion is to emphasize the fact that MicroMonitor is extremely flexible with regard to where it resides in the memory space of the target system. If you are already running with a ported version of the monitor, but you need to adjust the memory map, it's pretty simple to do it. The location of the monitor's TEXT space is generally determined by the CPU and its reset vector location. The placement of the monitor's BSS space is usually the base of ram but can exist anywhere, and is defined in the memory map linker file (through the variables in the makefile). These linker files are in the target specific directory and end with the dot suffix ".ld". They are built by make from the ".ldt" template file using variables defined in the makefile. In the case of monitors that copy themselves to RAM, an additional adjustment must be made in the makefile (UMON_START & UMON_RAMBASE discussed above). TFS can be configured in both flash and ram and is established by the config.h file and tfsdev.h (discussed below).

11.8 The Console (Serial) Driver and cpuio.c:

The first thing to do after the basic startup is to initialize the serial device. This should be done by the call to devInit() which is part of the init1() function in umon_main/target/common/start.c. The monitor supports a "device" based model for building drivers however, it's not used much, so it is likely that the devInit() function is going to be in cpuio.c and will simply initialize the serial port. Referring to cpuio.c under the template port directory, there are at least six functions that need to be populated for use by the monitor's console interface:

`int devInit (int baud)`	Initialize the console uart to the specified baud rate.
`int target_putchar (char c)`	Transfer the incoming character out the serial port.
`int target_getchar (void)`	Retrieve an assumed-to-be present character from the serial port.
`int target_gotachar (void)`	Return 1 if a character is present on the serial port; else 0.
`int target_console_empty (void)`	Return 1 if all characters have been flushed from the console output;

	else 0.
`void ConsoleBaudSet (int baud)`	Re-establish only the baud rate of the console serial port.

• **Table 3 : Console Driver Functions**

The first four functions listed in Table 3 are required for basic functionality of the console interface. The target_putchar(), target_getchar() and target_gotachar() functions are simple polled functions so should only require 2-3 lines of device-specific code. For getting started, target_console_empty() can simply return 1, but should eventually return 1 only after the console output is known to be flushed. Also ConsoleBaudSet() can be left empty, but should eventually re-define the console baud rate.

There are additional functions in cpuio.c that eventually need to be filled in (refer to umon_ports/template/cpuio.c for details); however, for the initial stages of the porting process, it is safe to leave them as they are in the template.

11.9 The Flash Driver:

At this point, you have a basic monitor working with a serial port. Next is flash. Start by setting INCLUDE_FLASH to 1 in config.h. There is one rule to be aware of regarding flash:

It is illegal to fetch instructions out of a flash device while operating on that flash device.

Note that there are exceptions to this rule; however, for the sake of this discussion we assume it to be a requirement. To test the functions we are about to discuss, you can use XMODEM to download small programs to RAM and run them by executing the "call" command, or if your monitor is built to run out of RAM, then you can use the pm & dm commands to peek and poke the flash address space to test out the basic algorithms. There are three fundamental functions needed for each flash device, and possibly a few additional functions, depending on the system:

- Flash_Init(): initialize the control structures used by the flash API.
- Flash_Erase(): erase a sector, or group of sectors.
- Flash_Write(): write any number of bytes starting at any point within a flash bank.

These first three are required for all systems to be able to read/write from flash, and hook the drivers to TFS. These additional functions support cases where there are multiple (different) devices, the ability to re-burn the monitor in-place (using newmon or Xmodem –B), and the ability to lock and/or unlock flash sectors (dependent on the capability of the flash device)…

- Flash_Type(): retrieve the manufacturer and device code.
- Flash_Ewrite(): erase and write in one step.
- Flash_Lock(): lock, unlock and lock query.

This text will not dig into any single device driver implementation. It is assumed that the reader has some working knowledge of the flash device. There are plenty of examples to start with, and chances are pretty good that the driver (or at least something close) already exists. The goal here is to describe in general terms the philosophy behind MicroMonitor's flash driver strategy. Based on that, following are a few discussions on various topics that the reader should be aware of…

11.9.1 What is a "*bank*" of flash?

The term "bank" here refers to a device or group of parallel devices that are accessed by a single CPU read or write. This may be one, two or four devices depending on the target configuration. If a single 8-bit boot flash device is in the system, that is a bank. If there is a set of 2 16-bit devices configured in parallel to form one 32-bit wide flash peripheral, that would also be one bank. Similarly, 2 or 4 8-bit devices in parallel could be a bank. Note that within any one bank, only one device type will exist; and when sector erase is performed, it is simultaneously performed on as many devices as are in parallel within that bank.

11.9.2 Flash operations copied to RAM

Elaborating a bit more on the "rule" stated in section 11.9 above, most flash devices do not support simultaneous operations. In other words, if you are fetching from the device, you can't erase (or write to) a

sector in that device. Some devices do allow you to fetch from one portion of the device while operating on the other portion; however, for simplicity this is ignored here. MicroMonitor simply assumes that the code that is running some flash operation must be in RAM (or in some other device). This is accomplished in one of two ways:

- If the monitor is built to fetch from flash at runtime, then the code that does the flash operations must be copied to RAM when the monitor is started.
- If the monitor copies itself to RAM for runtime execution, then by default, the code that does the flash operations is already in RAM so there is nothing further to do.

The tricky part is the first case. If a function is to be copied to a different location, then it must be relocatable or "position independent". This can be tricky, and definitely takes some additional thought on the part of the developer. The most important thing to avoid when building the relocatable function is to avoid calling any other function from within the relocated flash operation because, at best, that will put you back into non-relocated (hence flash) space. Also, you need to be aware of the compiler's ability to generate relocatable code.

The decision regarding whether or not the functions are copied to RAM is dealt with in `FlashInit()` using the function `flashopload()` and the #define FLASH_COPY_TO_RAM in config.h.

```
581: int
582: FlashInit(void)
583: {
584:     int     snum;
585:     struct  flashinfo *fbnk;
586:
587:     snum = 0;
588:     FlashCurrentBank = 0;
589:
590: #ifdef FLASH_COPY_TO_RAM
591:
592:     /* Copy functions to ram space... */
593:     /* Note that this MUST be done when cache is disabled to assure that */
594:     /* the RAM is occupied by the designated block of code. */
595:
596:     if (flashopload((ulong *)Intel28f640_16x1_lock,
597:         (ulong *)EndIntel28f640_16x1_lock,
598:         FlashLockFbuf,sizeof(FlashLockFbuf)) < 0)
599:         return(-1);
600:     if (flashopload((ulong *)Intel28f640_16x1_type,
601:         (ulong *)EndIntel28f640_16x1_type,
602:         FlashTypeFbuf,sizeof(FlashTypeFbuf)) < 0)
603:         return(-1);
604:     if (flashopload((ulong *)Intel28f640_16x1_erase,
605:         (ulong *)EndIntel28f640_16x1_erase,
606:         FlashEraseFbuf,sizeof(FlashEraseFbuf)) < 0)
607:         return(-1);
608:     if (flashopload((ulong *)Intel28f640_16x1_ewrite,
609:         (ulong *)EndIntel28f640_16x1_ewrite,
610:         FlashEwriteFbuf,sizeof(FlashEwriteFbuf)) < 0)
611:         return(-1);
612:     if (flashopload((ulong *)Intel28f640_16x1_write,
613:         (ulong *)EndIntel28f640_16x1_write,
614:         FlashWriteFbuf,sizeof(FlashWriteFbuf)) < 0)
615:         return(-1);
616:
617: #endif
618:
619:     fbnk = &FlashBank[0];
620:     fbnk->base = (unsigned char *)FLASH_BANK0_BASE_ADDR;
621:     fbnk->width = FLASH_BANK0_WIDTH;
622: #ifdef FLASH_COPY_TO_RAM
```

```
623:     fbnk->fltype = (int(*)())FlashTypeFbuf;     /* flashtype(). */
624:     fbnk->flerase = (int(*)())FlashEraseFbuf;   /* flasherase(). */
625:     fbnk->flwrite = (int(*)())FlashWriteFbuf;   /* flashwrite(). */
626:     fbnk->flewrite = (int(*)())FlashEwriteFbuf; /* flashewrite(). */
628:     fbnk->fllock = (int(*)())FlashLockFbuf;     /* flashelock(). */
630: #else
631:     fbnk->fltype = Intel28f640_16x1_type;
632:     fbnk->flerase = Intel28f640_16x1_erase;
633:     fbnk->flwrite = Intel28f640_16x1_write;
634:     fbnk->flewrite = Intel28f640_16x1_ewrite;
636:     fbnk->fllock = Intel28f640_16x1_lock;
638: #endif
639:
640:     snum += FlashBankInit(fbnk,snum);
641:
642:     sectorProtect(FLASH_PROTECT_RANGE,1);
643:
644: #ifdef FLASHRAM_BASE
645: #ifdef FLASHRAM_SECTORSIZE
646: #define ramSectors 0
647: #endif
648:     FlashRamInit(snum, FLASHRAM_SECTORCOUNT,
649:         &FlashBank[FLASHRAM_BANKNUM], sinfoRAM, ramSectors);
650: #endif
651:     return(0);
652: }
```

• Listing 47: FlashInit() Example

In most of the drivers already written, they support both modes through the #define FLASH_COPY_TO_RAM (see Listing 47, taken from umon_main/flash/devices/intel28f640_16x1.c). The important thing to note here is that if the functions are to be copied to RAM, then I&D caches should be off during this operation (or precaution should be taken to assure that the instructions copied to the RAM space are flushed from the data cache and the corresponding space is invalidated from the I-cache). The easiest way to deal with this is to initialize these drivers prior to enabling any cache functionality (that's what MicroMonitor does).

11.9.3 The flashinfo & sectorinfo structures

For each bank of flash, there is one entry in the FlashBanks[] array (a table of *flashinfo* structures) that must be initialized to provide the driver with basic information about the flash device[74]. This structure contains the high-level information needed by the driver… The address range occupied by the device, the device id, pointers to the functions that operate on the device, and a pointer to a table of *sectorinfo* structures. Referring to Listing 47, the code within lines 622-638 initialize the function pointers to either be the address at which they actually reside, or the address of the array into which they were copied.

```
struct sectorinfo {
       long   size;               /* size of sector */
       int    snum;               /* number of sector (amongst possibly */
                                  /* several devices) */
       int    protected;          /* if set, sector is protected by window */
       unsigned char *begin;      /* base address of sector */
       unsigned char *end;        /* end address of sector */
};

struct flashinfo {
       unsigned long id;          /* manufacturer & device id */
       unsigned char *base;       /* base address of device */
       unsigned char *end;        /* end address of device */
       int    sectorcnt;          /* number of sectors */
       int    width;              /* 1, 2, or 4 */
       int    (*fltype)();
```

[74] In config.h, the FLASHBANKS definition is used as the number of flash banks; hence, the size of the FlashBanks[] array.

```
        int     (*flerase)();
        int     (*flwrite)();
        int     (*flewrite)();
        int     (*fllock)();
        struct sectorinfo *sectors;
};
```

• Listing 48: The flashinfo and sectorinfo structures

This whole strategy assumes that the flash is broken up into sectors, so for each *flashinfo* structure, there is a table of *sectorinfo* structures that contain sector-specific information. This includes the size, beginning and end address, the sector number (as it exists in the system, not the single device) and the protection status of the sector[75]. There is one flashinfo structure for each bank and one sectorinfo structure for each sector in each bank. Note that if the bank is made up of a parallel set of devices, then the sector information must account for this. In other words, if the bank consists of two 29f040[76] devices in parallel, there is a total of 8 sectors in the bank and each sector is 128K (2x64K). The initialization of the flashinfo is done in FlashInit() (Listing 47) and sectorinfo structures is done in the `FlashBankInit()` () function at system startup.

```
510: /* This configuration is a single 28F640 device.
511:  * Each device has 64 128Kbyte sectors...
512:  */
513: int SectorSizes28F640_16[] = {
514:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
515:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
516:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
517:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
518:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
519:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
520:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
521:     0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
522: };
523:
524:
525: struct sectorinfo sinfo640[sizeof(SectorSizes28F640_16)/sizeof(int)];
526:
527: int
528: FlashBankInit(struct flashinfo *fbnk, int snum)
529: {
530:     uchar   *saddr;
531:     int     i, *sizetable, msize;
532:     struct  sectorinfo *sinfotbl;
533:
534:     /* Based on the flash bank ID returned, load a sector count and a */
535:     /* sector size-information table... */
536:     flashtype(fbnk);
537:     switch(fbnk->id) {
538:         case INTEL_28F640:
539:         case INTEL_DT28F640J5:
540:         case INTEL_DT28F128J5:  /* 128 device with only half addressable */
541:             fbnk->sectorcnt = (sizeof(SectorSizes28F640_16)/sizeof(int));
542:             sizetable = SectorSizes28F640_16;
543:             break;
544:         default:
545:             printf("Unrecognized flashid: 0x%08lx\n",fbnk->id);
546:             return(-1);
547:             break;
548:     }
549:
```

[75] MicroMonitor's flash interface provides a basic level of "soft" protection to each sector. If protected, then the sector can only be operated on if the flash protection window is opened. This is done with the "flash opw" command and remains opened for only the next command issued at the monitor CLI.

[76] The AM29F040 contains 8 sectors, each 64K

```
550:     /* Create the per-sector information table.  The size of the table */
551:     /* depends on the number of sectors in the device...  */
552:     if (fbnk->sectors)
553:         free((char *)fbnk->sectors);
554:     msize = fbnk->sectorcnt * (sizeof(struct sectorinfo));
555:     sinfotbl = (struct sectorinfo *)malloc(msize);
556:     if (!sinfotbl) {
557:         printf("Can't allocate space for flash sector information\n");
558:         return(-1);
559:     }
560:     fbnk->sectors = sinfotbl;
561:
562:     /* Using the above-determined sector count and size table, build */
563:     /* the sector information table as part of the flash-bank structure: */
564:     saddr = fbnk->base;
565:     for(i=0;i<fbnk->sectorcnt;i++) {
566:         fbnk->sectors[i].snum = snum+i;
567:         fbnk->sectors[i].size = sizetable[i];
568:         fbnk->sectors[i].begin = saddr;
569:         fbnk->sectors[i].end =
570:             fbnk->sectors[i].begin + fbnk->sectors[i].size - 1;
571:         fbnk->sectors[i].protected = 0;
572:         saddr += sizetable[i];
573:     }
574:     fbnk->end = saddr-1;
575:     return(fbnk->sectorcnt);
576: }
```

• Listing 49: The FlashBankInit() function

FlashBankInit() is called once for each flash bank. Actually, depending on the driver, the code that makes up FlashBankInit() may just be part of the FlashInit() function. If there is more than one type of flash device in the system, then there would be multiple sector-size tables. Usually there's only one device type, so there's only one table (line #513 of Listing 49). In cases where multiple different devices must be supported the code within lines 534-548 determine the device type. Then the loop within lines 565-573 initializes each entry in the sectorinfo table for that device. Notice that this strategy supports flash devices with different sector sizes by simply building a sector-size array that reflects the size of each sector within the device.

Two Driver "Styles":

As of Sept 2002, there are multiple "styles" that can be used to implement the flash drivers. The intent here is to not eliminate the original method for those who are comfortable there, but to provide a needed improvement in the way the drivers are structured. Note that this is simply a "style" issue. The interfaces are essentially identical, so either approach can be used. The two techniques are distinguished by the location in which the source is stored…

umon_main/target/flash/boards: This directory has multiple directories below it, with each directory dedicated to a single target hardware flash layout. The assumption is that you are building a driver that will work for your target's flash layout. If you have more than one device, then that directory would contain code for each device. If you want to support a "simulated" flash-in-RAM device for use by TFS, that detail would also be in that directory of code. So, for example, if I have one target that has a 29F040 boot flash and a 29LV160 flash for file storage, and another target with a 29F040 boot flash and a 28F128 for file storage, each directory would contain device interface code for their set of flash devices. The secure thing about this approach is that the directory will only be modified when a change is being made to that target; hence, you will have a platform on which you can test the driver changes. The inconvenient thing about this approach is that different target hardware may use the same device (the 29F040 in this example); however, the code must be duplicated for each target. This approach was the original approach for MicroMonitor because I did not want to have to worry about "adjusting" the driver for one target only to find out that the adjustment broke something for some other target.

Within each directory are at least three files. A file containing the code that would have to be relocated (usually contains "pic" in the name, pic=position-independent-code). A file that contains the basic establishment

of the flash control structures and a device header file. If more than one device is supported, then more files will exist.

umon_main/target/flash/devices: This directory has no subdirectories. The goal here is to provide a simple, single-device driver for targets that have only one flash device (which is probably the most common); but to also allow the target to include multiple files if the target has multiple flash devices. The point is that each file is specific to a particular device set up with a particular configuration (not a particular target hardware platform). The term "configuration" here refers to the width of the single device, the width of the overall bank and the number of devices put in parallel to form that bank. The naming convention for the device files is as follows (each device has a header file with the same name)...

```
DEVICENAME_DBW_DIP

where…
       DEVICENAME    is the device name (duh)
       DBW           is the configured device data bus width (in bits)
       DIP           is the number of devices in parallel (to form a bank)
```

Many flash devices can be configured in x8 or x16 bus width mode, so for example, the Am29LV160D can have several different source files depending on its configuration in the target system...

```
am29lv160d_08x1.c    -> Am29LV160D in  8-bit mode, 1 device  in parallel
am29lv160d_08x2.c    -> Am29LV160D in  8-bit mode, 2 devices in parallel
am29lv160d_08x4.c    -> Am29LV160D in  8-bit mode, 4 devices in parallel
am29lv160d_16x1.c    -> Am29LV160D in 16-bit mode, 1 device  in parallel
am29lv160d_16x2.c    -> Am29LV160D in 16-bit mode, 2 devices in parallel
```

Refer to the README file in the common/flashdev directory for more details. As of uMon1.0, the older "board" directory is not kept up-to-date (unless contributed), so it is recommended that new ports use the flash driver style found under the devices directory. A good starting point is a combination of a known-working driver under devices and the flash_template.c file also in that directory.

11.9.4 Description of FLASH configuration parameters in config.h...

Note that the use of these definitions depends on how you have configured your flash driver, so actual naming may be slightly different.

FLASHBANKS	Set this to the number of different banks of flash your target system will have.
FLASH_BANK0_WIDTH	Set to the width of flash bank zero. This is in 8-bit increments, so a 16-bit bank has a width of 2.
FLASH_BANK0_BASE_ADDR	Base address of flash bank zero.
FLASH_LARGEST_SECTOR	Size of the largest sector in all of the flash banks
FLASH_PROTECT_SIZE	This is a definition used in some of the early flash drivers. This value would be set to the highest location in flash that needs software protection. The assumption was that all sectors occupying the space between 0 and this value would be protected. This doesn't work well with targets that do not have their boot flash at address 0, so it has been replaced with FLASH_PROTECT_RANGE. If you are writing a new flash driver, use FLASH_PROTECT_RANGE instead of this.
FLASH_PROTECT_RANGE	This is some string formatted as individual, comma-delimited or dash-delimited ranges. A few valid choices would be "0", "0-3", "0-3,8". Protected flash simply means that the flash command will require preliminary "opw" (open window) request for those banks to be modified. See the flash command for more details. This is a replacement for FLASH_PROTECT_SIZE. If you are writing a new flash driver, use this instead of FLASH_PROTECT_SIZE.
FLASH_COPY_TO_RAM	If defined, then the flash driver functions are copied to RAM and run through function pointers that point to the blocks of RAM allocated for

	the relocated flash drivers.
LOCK_FLASH_PROTECT_RANGE	If defined, then the range of sectors specified by FLASH_PROTECT_RANGE will not only be software protected, but they will also (assuming the hardware supports it) be locked at startup.
WIDTHX	This define is only used with flash drivers that are written to be configurable for different widths simply by establishing this macro. Currently, the only device that supports this in the monitor code is the 29F040. Refer to that header file for details.

Note that if there was more than one flash bank in your target system, then there would be an additional set of FLASH_BANKN_XXX macros for each additional bank.

Once the flash driver is operational, the Xmodem -B command can be used to download new monitor binaries without the need for an external programmer.

11.10 Configuring TFS

Once the flash drivers are written and tested, then adding TFS is a trivial next step. The first thing to do here is to determine how much of TFS you want to include in your port. This is usually only a consideration for systems that have a limited amount of flash that can be allocated to the monitor binary.

INCLUDE_TFS	This is the core of TFS functionality. It is a minimum-configuration and a prerequisite to each of the following options. Files are managed, and based on the presence of files, the autoboot functionality is in place. The basic ability to add and delete files using tfsadd & tfsunlink is provided (this allows files to be transferred if TFTP and/or XMODEM is enabled). Plus, an application has the ability to hook to those functions.
INCLUDE_TFSAPI	This pulls in the code that allows the user to call the functions that make up the extended TFS API: tfstruncate, tfseof, tfsread, tfswrite, tfsseek, tfsgetline, tfsipmod, tfsopen, tfsclose.
INCLUDE_TFSCLI	This pulls in the code that is responsible for the “tfs” command on the command line interface.
INCLUDE_TFSSCRIPT	This pulls in the code and commands that support scripting. Some of the commands in the CLI (i.e. if, exit, return, gosub, goto, read, etc…) are only applicable from within a script; hence, they are only pulled into the monitor build if TFS scripting is enabled. Refer to Chapter 7 above for more details on scripting within MicroMonitor.
INCLUDE_TFSSYMTBL	This pulls in the code that is used by MicroMonitor to support the ability to deal with symbols on the command line. Refer to section 10.3.1 above for details on the use of symbols within MicroMonitor.

Based on the above descriptions, set the above entries to 1 or 0. As a rule of thumb, if you have the memory, then just set the all to 1. The next set of definitions are used to properly place TFS within the memory space of the target…

TFSSPARESIZE	This value should be set to the size of the largest flash sector that is dedicated to TFS
TFSSECTORCOUNT	This should be set to the number of sectors that are to be dedicated to TFS file storage space. This does not include the SPARE sector.
TFSSTART	Base address of the beginning of TFS.
TFSEND	End address of TFS file storage space that started at TFSSTART. Note that this address should NOT include the space used by the spare sector. Usually, TFSEND is 1 byte less than TFSSPARE because the spare sector lies just after the end of TFS storage space.
TFSSPARE	Base address of the spare sector that is to be used by the TFS

	storage space starting at TFSSTART.
TFS_DISABLE_MAKE_BEFORE_BREAK	If defined, then TFS's default "safe" behaviour to not remove a file until it's copy is in place is turned off. This is useful for systems where the size of a file being stored is larger than half the size of all TFS space. As a result, when a file is copied over an exising file of the same name, the existing file is deleted prior to installing the new file.
TFS_DISABLE_AUTODEFRAG	If defined in config.h, automatic defragmentation will not be done. It will be up to the user to run "tfs clean" from the command line when necessary.
TFS_VERBOSE_STARTUP	Set this to 1 if you want TFS to print out a message at startup indicating that it is configuring each TFS device. This is usually only applicable if the device size is large, and provides a bit of a sanity check at startup.
TFS_AUTOBOOT_ ABORTABLE	Pre Feb 2004, the automatic execution of monrc and/or non-query autoboot files was not abortable under any circumstance. Post Feb 2004, autoboot of these types of files can be aborted if this macro is defined. See section 11.10.1 for a discussion.
AUTOBOOT_ABORT_CHAR	This defines the character that is used to invoke the autoboot abort (if it is enabled). The default character is 0x03 (ctrl-c).
TFS_AUTOBOOT_CANCEL_CHAR	If defined, this is the required character used as input to abort a single queryable autoboot file. If not defined, then any character is accepted.
TFS_ALTDEVTBL_BASE	If defined, this is part of the code needed to support the ability to reconfigure TFS's use of flash space at runtime.
TFS_RUN_DISABLE	If set to non-zero, it will build the monitor with the "tfsrun" facility disabled. The primary use of this macro is to allow a monitor image to be built and installed into a target's RAM space to recover from a corrupted monrc or autobootable file.
TFS_EBIN_ELF TFS_EBIN_COFF TFS_EBIN_AOUT TFS_EBIN_MSBIN TFS_EBIN_ELFMSBIN	If there is a need to use TFS's loader, then one (and only one) of these definitions should be set. The choice here depends on the cross compilation toolset being used to build the application code.

There is also tfsdev.h. This header file is used to help the TFS code with targets that have more than one memory device over which TFS will span. For most targets, TFS will only span one contiguous block of flash; however, TFS does support multiple non-contiguous blocks. The tfsdev.h structure establishes this. So for most targets, the structure will look like this...

```
struct tfsdev tfsdevtbl[] = {
{
       "//FLASH/",
       TFSSTART,
       TFSEND,
       TFSSPARE,
       TFSSPARESIZE,
       TFSSECTORCOUNT,
       TFS_DEVTYPE_FLASH },

{ 0, TFSEOT, 0, 0, 0, 0, 0 }

};
#define TFSDEVTOT ((sizeof(tfsdevtbl))/(sizeof(struct tfsdev)))
```

The first member of the first entry in the table is the name to be associated with the device. The majority of the remaining members of the structure specify the starting point, ending point and sector info for the

device. TFS can be set up so that the device used is somewhat dynamic (within a family). If you have a target that may be configured with a 29F040 or a 29F010 for example, then set the TFS_DEVTYPE_DYNAMIC bit and all other entries NULL. This assumes that you have written your flash driver to deal with this.

```
struct tfsdev tfsdevtbl[] = {

{
        "//FLASH/",
        TFSSTART,
        TFSEND,
        TFSSPARE,
        TFSSPARESIZE,
        TFSSECTORCOUNT,
        TFS_DEVTYPE_FLASH | TFS_DEVINFO_DYNAMIC },

{ 0, TFSEOT, 0, 0, 0, 0, 0 }

};
```

If there is a need to configure a second device, then the parameters for that second entry would be taken from a second set of parameters that would be established for the second device in config.h (like TFSSTART_1, TFSEND_1, TFSSPARE_1, etc...). The structure would then simply have a third entry...

```
struct tfsdev tfsdevtbl[] = {
{
        "//F0/",
        TFSSTART,
        TFSEND,
        TFSSPARE,
        TFSSPARESIZE,
        TFSSECTORCOUNT,
        TFS_DEVTYPE_FLASH | TFS_DEVINFO_DYNAMIC },
{
        "//F1/",
        TFSSTART_1,
        TFSEND_1,
        TFSSPARE_1,
        TFSSPARESIZE_1,
        TFSSECTORCOUNT_1,
        TFS_DEVTYPE_FLASH | TFS_DEVINFO_DYNAMIC },

{ 0, TFSEOT, 0, 0, 0, 0, 0 }

};
```

Once TFS has been configured and built into the monitor, the command “tfs stat” should be run so that you can observe what TFS thinks it is configured for. Also, issue the command “tfs clean” so that TFS can do a first-time defragmentation just to verify that its configuration is valid.

11.10.1 Abortable Non-query Autobootable Files

Until around Feb 2004, the automatic execution of the monrc file and any non-query autobootable file was not abortable in any way (without external hardware). This maintains a level of startup security so that an application can assume that if it is configured to autoboot without a query, then it will. This is good, except for those cases when the non-queriable autobootable file is messed up and it causes the target to hang; hence resulting in the need to resort to external equipment to re-burn the boot flash. This is something that is very likely to happen during project development, so to get around this, TFS now supports the ability to abort these files if configured to do so. The macro TFS_AUTOBOOT_ABORTABLE can be defined to enable this option. If, this #define is set, then when the autoboot process runs during MicroMonitor startup, the abort character

(defined by AUTOBOOT_ABORT_CHAR, or default of 0x03) can be typed at the console to abort the autoboot. To still support security, the abort process looks to see if there is a password file installed in TFS and if there is one, then the abort will only succeed if the correct password is used. If there is no password, then there is apparently no need for security, so the abort succeeds automatically.

Note that this should not be confused with TFS_AUTOBOOT_CANCEL_CHAR which is a character that can be defined as the character that must be typed when a file autoboots with query. Without this definition in config.h, the default action of MicroMonitor is to accept any character as the cancel character.

11.10.2 Power Safe Defragmentation (or not)

TFS can be configured to run with or without power-safe defragmentation. The obvious advantages of using the power-safe mode is that if a defrag is in progress when a power hit or reset occurs, it will recover. The disadvantage is that this defragmentation takes up time and code. If power safe defrag is used, then the file tfsclean1.c is part of the build, and a spare flash sector is used for temporary storage as each sector is defragmented. If power safe defrag is not used (tfsclean2.c), then TFS must be able to assume that starting at the APPRAMBASE location there is a block of RAM that is at least as large as the flash space to be defragmented. In this mode, at the time of a defrag, the valid files are copied to RAM, then the flash is erased and files are copied back to flash. This process is obviously much quicker, but is very vulnerable to a reset or power hit.

11.10.3 Allocating a Section of NVRAM as a TFS Device

Before starting this topic, a few points need to be made:

- This topic is not really part of the basic porting process however, it is something that may be useful in your target; hence, after completion of the port, this may be a nice enhancement. The point is that these additional TFS devices should be configured in after the initial, flash-based TFS section of the port has been completed and verified. For the basic porting process, just configure the simplest TFS configuration.
- A few changes (circa Jan 2004) have been made to the flash common code to support a smoother mechanism for adding an NVRAM based TFS area. If you are using code prior to Jan 2004, make sure you get updated.
- As of uMon 1.0, there is a distinction between NVRAM and RAM based TFS devices. Prior to uMon 1.0, any RAM device (volatile or non-volatile) required an entry in the tfsdev.h structure; hence, it was created at build time. As of uMon 1.0, a simple, volatile RAM-based device can be configured at runtime with the new tfs command “ramdev”. This is assumed to reside in volatile ram that will be cleared at reset. The NVRAM device type is applicable to RAM that is battery-backed; hence, can recover from a power cycle or reset[77].
- This discussion assumes you are using the newer flash drivers (under flashdev); however, it is applicable to both.

In most situations, TFS is used to store files in non-volatile flash memory space. This makes sense. Files are usually established such that, when the system is powered up, it can access files immediately out of on-board flash. However, there are cases where it would be real nice to have some block of RAM space that can be allocated to TFS storage as well... Two types are available: RAM (volatile) and NVRAM (possibly non-volatile). The simple RAM type can be created at runtime with the “tfs ramdev” command. The NVRAM type (which may or may not actually be using non-volatile RAM) is supported through the use of a FLASHRAM device configured at build time…

To support this, the flash drivers and TFS can be configured so that an area of RAM space on the target simulates a block (set of sectors) of flash. A block of RAM is allocated to the flash drivers and the flash driver is aware of the flash device type "FLASHRAM". With the flash drivers being aware of this, the tfsdev.h file is configured to use this space as a storage device. There are three affected files: flashdev.c, config.h and tfsdev.h...

[77] The NVRAM device type can still be applied to volatile RAM (beware of the volatility); however, with the new “ramdev” command in TFS this isn’t needed.

Additions to config.h in the target-specific directory:

The config.h file must have some of the basic configuration information to be used by tfsdev.h, flashdev.c and flash.c, so initially a few decisions have to be made:

- Where is the RAM-based TFS bank to be located?
- How big should the bank be?
- Is it ok to use one sector size?

The first two questions are rather obvious, the answer to the third question determines whether or not default common code can be used or a target-specific table needs to be established[78]. The common code used for flash RAM bank setup is only pulled in if FLASHRAM_BASE is defined, so the config.h file should be set up such that the definition of FLASHRAM_BASE pulls in all the associated definitions, plus it should be used to increment the number of flash banks. This approach is handy just to support the possibility that the flash RAM bank will not be permanent; hence removal of that #define will be all that is needed to remove it from the configuration. Refer to the following definitions for an example configuration, note that these values are valid for the Cogent CSB360 target.

```
#define FLASHRAM_BASE                0x380000
#ifdef FLASHRAM_BASE
# define FLASHRAM_END                0x3fffff
# define FLASHRAM_SECTORSIZE         0x010000
# define FLASHRAM_SPARESIZE          FLASHRAM_SECTORSIZE
# define FLASHRAM_SECTORCOUNT        8
# define FLASHRAM_BANKNUM            1
# define FLASHBANKS                  2
#else
# define FLASHBANKS                  1
#endif
```

FLASHRAM_BASE	Base address of the block of RAM allocated to “fake” flash
FLASHRAM_END	Last writeable address in the block
FLASHRAM_SECTORSIZE	Size of each “fake” sector (if applicable).
FLASHRAM_SPARESIZE	Size of the spare sector (must be the largest in the block).
FLASHRAM_SECTORCOUNT	Number of sectors in the block.
FLASHRAM_BANKNUM	Index into the FlashBanks[] array (starts with 0) that is to be used by the FLASHRAM.
DISABLE_INTERRUPTS_DURING_FLASHOPS	If defined, interrupts are disabled during flash operations.

Note that with the above set of definitions, the value of (FLASHRAM_SECTORCOUNT * FLASHRAM_SECTORSIZE) should be equal to (FLASHRAM_END - FLASHRAM_BASE + 1). Also, the definition of FLASHRAM_SECTORSIZE implies that all sectors must be the same size. By default this is assumed; however, it is not required (as you will see below).

Additions to tfsdev.h in the target-specific directory:

We are adding an additional device to the TFS device list, so we need to add an entry to the tfsdev.h structure table. The device type must be set to TFS_DEVTYPE_NVRAM. The definition of TFS_DEVINFO_AUTOINIT allows the user to configure an automatic call to memset() that will set the entire block of RAM to 0xff at startup. This assures that the RAM space looks like erased flash at startup, and may or may not be applicable (if RAM is battery backed, then you may want to assume that the space is non-volatile; hence no need to init at startup).

```
#ifdef FLASHRAM_BASE
```

[78] Usually the answer is yes; however, there may be cases where a small block of battery backed RAM is being used and a common sector size may not be appropriate.

```
{
       "//RAM/",
       FLASHRAM_BASE,
       FLASHRAM_END - FLASHRAM_SPARESIZE,
       FLASHRAM_END - FLASHRAM_SPARESIZE + 1,
       FLASHRAM_SPARESIZE,
       FLASHRAM_SECTORCOUNT-1,
       TFS_DEVTYPE_NVRAM | TFS_DEVINFO_AUTOINIT },
#endif
```

The definition of each of these entries was discussed in the section above. This entire structure is wrapped with #ifdef FLASHRAM_BASE so that it can easily be configured out of the system when/if necessary.

Additions to flashdev.c in the flash-specific driver code:

At the bottom of the FlashInit() function, one additional flash bank must be configured. The bulk of the code is in the monitor's common flash file common/monitor/flash.c, in the function FlashRamInit(). In FlashInit() (which is part of the flash-specific driver code), a call to FlashRamInit() must be added. If the sector sizes will vary, then an initialized array that specifies the size of each sector to be configured in the RAM bank must be allocated[79]. An example of the tables needed for this follows, and note that it is only necessary when the sector size will vary. If the sector size is constant, then the FlashRamInit() function will use the defines in config.h (FLASHRAM_SECTORSIZE and FLASHRAM_SECTORCOUNT) to build this array automatically. The size and number of sectors depends on the amount of space that is allocated to this bank.

```
#ifdef FLASHRAM_BASE
#ifndef FLASHRAM_SECTORSIZE
int
ramSectors[] = {
    0x10000, 0x10000, 0x10000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000,
    0x10000, 0x10000, 0x10000, 0x20000, 0x20000, 0x20000, 0x20000, 0x20000
};
#endif
#endif
```

And, finally, at the bottom of FlashInit() the call to FlashRamInit() would be as follows. Note that the actual number of banks depends on the target configuration. In the case below, there is one real FLASH bank and one RAM bank emulating FLASH.

```
#ifdef FLASHRAM_BASE
#ifdef FLASHRAM_SECTORSIZE
#define ramSectors 0
#endif
       FlashRamInit(snum, FLASHRAM_SECTORCOUNT,
              &FlashBank[FLASHRAM_BANKNUM], sinfoRAM, ramSectors);
#endif
```

To verify that the new TFS device is installed correctly, the command “tfs stat” will dump the configuration. In the case of the Cogent CSB360 with FLASHRAM_BASE defined, the output of tfs stat will look something like this (note the additional //RAM/ device in boldface below)…

```
uMON>tfs stat
TFS Memory Usage...
      name    start       end       spare     spsize  scnt type
  //FLASH/: 0xff880000|0xfffdffff|0xfffe0000|0x020000|  59|0x200000
    //RAM/: 0x00380000|0x003effff|0x003f0000|0x010000|   7|0x140000

 Total memory: 8388608 bytes (used=1202742, avail=7185866 (7185554 for data)).
 Per-device overhead: 196904 bytes (defrag-state=296 spare-sector=196608).
 File data space: 1002078 bytes (live=43599, dead=958479).
```

[79] Note that the RAM doesn't have sectors; however we are configuring the RAM to look like flash, so we have to fake it here

```
 File overhead space: 3760 bytes (live=1092, dead=2668).
 File count: 36 (live=7, dead=29).
 Defrag will release 961147 bytes

TFS Hdr size: 92
Total files currently opened: 0
uMON>
```

11.10.4 Setting Up TFS's Alternate Device Table

This is one of those features that you don't want to add to your port until after you've got the basic stuff working. It's not complicated, but its just another tid bit that can wait till your port is stable. That being said, as of uMon 1.8, TFS supports the ability to repeatedly reconfigure the flash space allocated to a particular partition (or device) of TFS. If you're interested in having this capability, read on; else just remove the TFS_ALTDEVTBL_BASE definition from your port's config.h file, and skip to the next section of text. If the TFS_ALTDEVTBL_BASE definition is defined in config.h, then the common code (i.e. umon_main/target/common/tfs.c) assumes that it points to a "motionless" area in flash. This is similar to "moncomptr" and "etheraddr", simply meaning that from one build of uMon to the next, the location of this structure will not change. To implement this, four things must be added to the port specfic code…

Initially, the TFS_ALTDEVTBL_BASE definition should be commented out (or removed) from the config.h file. This allows you to establish the new block of flash space to be used by the alternate TFS device table, then install that update onto the board and run a few "dm" commands to make sure things are located in physical space where you think they should be located.

First, a block of flash must be allocated to the alternate TFS device table structure. This block of flash should be initialized to the erased state (0xff) and must be equal (or larger) in size to the default TFS device table defined in tfsdev.h for each port. The file umon_main/target/common/alttfsdevtbl.S provides a basic example and may be included in your port's flash-resident boot code (usually this is the rom_reset.S file). If your TFS device structure in tfsdev.h is larger than normal, you may need to create your own block by simply pulling that file's source into your rom_reset.S and adding to the size of the initialized data. In most cases the structure in tfsdev.h is only 2 or 3 entries, so alttfsdevtbl.S will work fine. Refer to umon_ports/template/reset.S for an example of including this file in the reset assembler code.

Second, all non-flash based versions of the monitor built for your port must reference this address in their linker map file. This simply means that the address at which the flash space is allocated must be reflected in the linker map file with a line similar to this… `alt_tfsdevtbl_base = ALTTFSDEVTBLBASE;` . Refer to umon_ports/template/TEMPLATE_ramtst.ldt for an example.

Third, the makefile must contain the entries for configuring the ram-based linker map files with the above mentioned entries. The .ldt files are templates that are used to construct the .ld files; hence the `ALTTFSDEVTBLBASE` value mentioned above must be replaced with the hard-coded address that is specific to your port. This is done in the makefile (refer to umon_ports/template/makefile) with added arguments passed to the `$(MAKE_LDFILE)` tool (vsub). At the top of the makefile is the variable replacement (i.e. `ALTTFSDEVTBLBASE=0xfff80020`[80]) and below, for each ram-based build, the `$(MAKE_LDFILE)` line must include `ALTTFSDEVTBLBASE=$(ALTTFSDEVTBLBASE)`.

Upon completion of this third step, the monitor should be built (note that we still haven't defined TFS_ALTDEVTBL_BASE in config.h) and installed on-target. Then, use the "dm" command just to look at the memory space at which the alt_tfsdevtbl_base block of flash has been placed. If that empty block of flash is where you configured it to be in the makefile, then you can go ahead and add the #define to config.h and rebuild one last time. If things don't appear to be lined up, then either adjust the rom_reset.S file containing the alttfsdevtbl.S block or adjust the address set up for ALTTFSDEVTBLBASE in the makefile. Then rebuild and install. It's important to do this check so that the ram and/or ramtst version of the monitor build finds the flash block.

[80] Obviously the address used here is port-specific.

With the above changes made to your port-specific code, uMon's TFS is now capable of dealing with multiple TFS reconfigurations, plus it will work just fine for systems that relocate themselves to run out of RAM. As of this writing, the TEMPLATE, CSB472, CSB536, CSB625, CSB650, CSB655 and CSB637 ports have been updated and tested to support this, so refer to them for example implementations.

11.11 The Watchdog Macro

Some target systems have a built-in watchdog of some kind and as a result, the code must be able to tickle the watchdog periodically just to indicate that everything is running smoothly. In the case of MicroMonitor, the WATCHDOG_MACRO can be defined in config.h. Usually this is simply a function that toggles a bit or does whatever is necessary to keep the watchdog hardware happy. The macro is inserted in strategic points in the code that may "take a while". One important requirement of the WATCHDOG_MACRO is that it be inline; hence making it insertable into the relocatable flash operations. Following is an example of a defined, inline watchdog macro in config.h…

```
#define WATCHDOG_MACRO \
        { \
        *(unsigned long *)WATCHDOG_ADDRESS |= WATCHDOG_BIT; \
        *(unsigned long *)WATCHDOG_ADDRESS &= ~WATCHDOG_BIT; \
        }
```

With this established in config.h, and assuming the WATCHDOG_MACRO is properly inserted in the flash driver[81], the target running uMon will effectively maintain a watchdog whose timeout period is greater than about a second.

11.12 Miscellaneous entries in config.h.

DEFAULT_ETHERADD	If all else fails, this will be the Ethernet address used by the monitor. If not specified in config.h, then it defaults to 00:00:00:00:00:00 in ether.h and the Ethernet interface is not turned up at runtime.
DEFAULT_IPADD	If all else fails, this will be the IP address used by the monitor. If not specified in config.h, then ether.h will assign 0.0.0.0.
APPRAMBASE_OVERRIDE	By default, the monitor will automatically establish a base address at which the application code can start assuming ownership of. This is done by simply finding the next 0x1000 modulo address after the end of the monitor's .bss section. This address is loaded into the APPRAMBASE shell variable and used by both the monitor and user for various things. If there is a reason to override this default setting, then put the override value in this definition; otherwise, this can be omitted.
BOOTROMBASE_OVERRIDE	Similar to APPRAMBASE_OVERRIDE, this value is automatically set to what the monitor sees as its base FLASH address. An example of the need for this is when the monitor is built to run out of RAM. The RAM-resident image needs an override so that the real boot rom base address is specified. If for some reason this needs to be overridden, then it should be loaded into this macro.
CPU_LE	Define this if the CPU is little endian.
FORCE_BSS_INIT	When the monitor first starts up, if it is a warmstart, then the .bss space of the monitor is not initialized to zero so that certain state variables (and shell variables) will survive a warmstart. If this is not desirable, then set this flag and all of the monitor's restart options will look like a coldstart.
LOOPS_PER_SECOND	Since the monitor does not use any interrupts, all timing is based on loop counts. To get this at-least close to being accurate, this define

[81] Some flash drivers may not have this installed simply because this is new as of uMon 1.0.

	should be set to the appropriate value. Use the sleep -c option to determine what the optimum setting is for this define.
SYMFILE	This define establishes the filename used by the monitor as the symbol-table file. Default is “symtbl”.
CPU_NAME	String that contains the name of the CPU. This is displayed as part of the boot-up header.
PLATFORM_NAME	String that contains the name of the target platform. This is displayed as part of the boot-up header and is also stored in the PLATFORM shell variable.
ALLOCSIZE	The amount of memory space that is to be allocated to the monitor's own heap. The heap in the monitor does not simply grow up from the end of .bss. It is a statically allocated array within the monitor's own .bss space. This avoids a conflict with other applications that may want to use the memory in some other way.
MONSTACKSIZE	This is the size of the monitor stack which is simply an array (MonStack[]) defined in umon_main/target/common/start.c.
TOD_IN_MONHEADER	If defined, then the function showDate() will be called at the bottom of the output of the monitor header. This function is assumed to print the current time of day.
USR_HEADER[1-4]	Four additional lines that can be added to the monitor header output to the console at startup.
MALLOC_DEBUG	Used in malloc.c to convert malloc(int) to malloc(int, __FILE__,__LINE__). This data is part of the output of a heap dump; hence, can be used to debug a memory leak in both the monitor itself and applications.
TARGET_ENV_SETUP	Used in env.c, this macro can be defined as a function name so that target-specific environment variables can be established automatically at startup.
MON_CMDLIST_HEADER	If defined, this is assumed to be a string (used by the "help" command) that contains text that will be output at the top of the portion of the command list that displays the monitor's built-in commands.
INCLUDE_VERBOSEHELP	This is usually defined to 1; however, to save space, but reduce the verbosity of the help output per command, set this to 0.
APP_CMDLIST_HEADER	If defined, this is assumed to be a string (used by the "help" command) that contains text that will be output at the top of the portion of the command list that displays the commands that have been installed by the application (using mon_addcommand()).
PRE_COMMANDLOOP_HOOK	Define this as a function that will be called just prior to uMon entering the main command processing loop.
PRE_TFSAUTOBOOT_HOOK	Define this as a function that will be called just prior to uMon running the TFS autoboot code.
DONT_CENTER_MONHEADER	If defined, then this simply disables the monitor's default action to center the banner that the monitor dumps to the console after a reset.

11.13 Adding a Target-Specific Command

Within MicroMonitor there are dozens of standard commands that apply to all targets. The command table can be found in the file common/monitor/cmdtbl.c. For many ports this standard command set is adequate; however, there are times that some target-specific attribute justifies adding new commands to the monitor’s command list. The simplest thing to do would be to just insert the new command into the command table; however, then the common code of the monitor is no longer common. To get around this, MicroMonitor’s command table has provisions for adding external commands. The command table is a table of structures…

```
#include "xcmddcl.h"

...
```

```
struct monCommand cmdlist[] = {
#if INCLUDE_ETHERNET
       { "arp",             Arp,          ArpHelp,      0},

...

       { "version",  Version,      VersionHelp},

#include "xcmdtbl.h"                      /* For non-generic commands that are */
                                          /* specific to a particular target.  */
       { 0,0,0,0 },
};
```

• Listing 50 : MicroMonitor Command Table

Many of the entries in this table are wrapped with some #if INCLUDE_XXX preprocessor directive because the command can be included or not included depending on the state if various INCLUDE_XXX definitions in the config.h file discussed earlier. Just above the bottom NULL entry of the list, is the provision that allows the monitor's command table to be extended without touching the common code.

```
{"date",             date,         dateHelp,     0},
{"i2c",              i2c,          i2cHelp,      0},
```

• Listing 51: Example Content of xcmdtbl.h

```
extern int date();
extern char *dateHelp[];

extern int    i2c();
extern  char *i2cHelp[];
```

• Listing 52: Example Content of xcmddcl.h

The files xcmdtbl.h (external command table) and xcmddcl.h (external command declarations) must be included in all MicroMonitor builds even if no additional commands are included. The above listings are those used in the Cogent CSB360 to add a "date" command and "i2c" command. Obviously, this requires that there also be the additional code that runs these commands, but that code would just be in another file included in the MicroMonitor build.

11.14 Post-Port Testing

It's easy to "almost" complete a port, and not realize that it's incomplete until something doesn't work. So, while your head is still in the porting process, it's wise to step back and run a few general sanity tests to make sure that everything is done. Some of these tests are functional steps that exercise specific sections of the monitor's feature set and others are simple "did you remember to do this" suggestions…

11.14.1 Test: Commands "reset" vs "reset –x"

If the state handoff from warmstart (in assembler) to start() (in C) is done properly, then there is a distinct difference between the commands "reset" and "reset –x". The "reset" command is supposed to attempt to simulate a hardware reset as close as possible. Actually, there is a target_reset() function that is called at this point to allow the firmware to invoke whatever the hardware provides to come as close to a hardware-reset as possible. This isn't always that easy, so in some cases, target_reset() (in cpuio.c) simply calls coldstart or warmstart(INITIALIZE). The bottom line is that "reset" should cause all autoboot files to be re-run and all the monitor's BSS space to be re-initialized. It will cause the monitor's startup header to be displayed, and all shell variables that are not part of autoboot are cleared as a result of the BSS reinitialization. The "reset –x" command is intended to simulate an application exit. In this case, the autoboot should not occur, the environment is left in tact and the message: "Application Exit Status: 0" is displayed. If both of these commands appear to reset the target completely, then it is likely that the config.h file still has FORCE_BSS_INIT set to 1. Now would be a good time to clear that, to verify that you're state handoff from reset to start() is clean.

11.14.2 Test: Flash-to-Flash Write

Verify that the “flash write” command can successfully write from one block of flash to another.

11.14.3 Test: Unaligned Flash Accesses

When running on a target that has a flash device that is wider than 8-bits (one byte), then it is wise to verify a that the driver can successfully write 1,2,3 & 4 bytes starting at 4 different locations whose 2 least signficant bits of address are 0, 1, 2 & 3. The fact that the previous test for TFS File Creation succeeded indicates that the flash driver is at least in pretty good shape. To extensively test the driver’s ability to write various data sizes to both aligned and unaligned addresses takes a bit more testing. This test is handled with a script that comes with the template port called flashtest.scr. It requires that INCLUDE_EE be set to 1 in config.h and to run properly, it assumes that the shell variable SECTORBASE is loaded with the base address of some sector in TFS space that can be written to during the test. It also assumes that the shell variable SNUM is loaded with the sector number whose base address is $SECTORBASE. With the values of SECTORBASE and SNUM being target-specific, the output of a successful run would look something like…

```
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a 2a 2a 2a 2a 2a 2a   2a 2a 2a 2a 2a 2a 2a 2a   ****************
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: ff 2a 2a 2a 2a 2a 2a 2a   2a 2a 2a 2a 2a 2a 2a ff   .**************.
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: ff ff 2a 2a 2a 2a 2a 2a   2a 2a 2a 2a 2a 2a ff ff   ..************..
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: ff ff ff 2a 2a 2a 2a 2a   2a 2a 2a 2a 2a ff ff ff   ...**********...
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: ff ff ff ff 2a 2a 2a 2a   2a 2a 2a 2a ff ff ff ff   ....********....
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: ff ff ff ff ff 2a 2a 2a   2a 2a 2a ff ff ff ff ff   .....******.....
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: ff ff ff ff ff ff 2a 2a   2a 2a ff ff ff ff ff ff   ......****......
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: ff ff ff ff ff ff ff 2a   2a ff ff ff ff ff ff ff   .......**.......
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a 2a 2a 2a 2a 2a 2a   ff ff ff ff ff ff ff ff   ********........
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a 2a 2a 2a 2a 2a ff   ff ff ff ff ff ff ff ff   *******.........
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a 2a 2a 2a 2a ff ff   ff ff ff ff ff ff ff ff   ******..........
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a 2a 2a 2a ff ff ff   ff ff ff ff ff ff ff ff   *****...........
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a 2a 2a ff ff ff ff   ff ff ff ff ff ff ff ff   ****............
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a 2a ff ff ff ff ff   ff ff ff ff ff ff ff ff   ***.............
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a 2a ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   **..............
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
20700000: 2a ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   *...............
20700000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
```

• Listing 53: Flash Test Script Output

Notice that every other line of output is just verification of the flash erasure. The script exercises the flash driver’s ability to write all combinations of aligned and unaligned sizes, starting addresses and ending addresses. This script can be found in the template directory in the file named “flashtest.scr”.

11.14.4 Test: TFS File Creation

Assuming TFS is enabled in your configuration, then run the command “tfs add testfile $APPRAMBASE 64”. Follow that with “tfs ls” and verify that the files “testfile” has been created and its size is 64 bytes. Then run “tfs rm testfile” and once again follow that with “tfs ls” to verify that the file has been deleted.

11.14.5 Test: Run "tfs clean"

The "tfs clean" command is used to force a TFS defragmentation. When it starts up it does some sanity checking on the structures established in tfsdev.h. If the tfsdev.h header is ok, then the command will run to completion. If there is any inconsistency in the tfsdev.h structure, this will be detected.

11.14.6 Test: Verify "tfs cfg"

If the TFS_ALTDEVTBL_BASE macro is defined in config.h, then the "tfs cfg" command should be useable in flash-resident and ram-resident versions of uMon.

11.14.7 Test: Baud Rate Changes

Use the "-b" option of the "set" command to change the console baud rate. The baud rates that are supported are target dependent, but at a minimum, 9600, 19200, 38400 & 57600 should work just fine. Note that each time you change the baud rate on the target, you need to change the setting of your terminal emulator (e.g. Hyperterminal) as well.

A second test for this is to set the CONSOLEBAUD shell variable in the monrc file and verify that the baud rate changes on reset.

11.14.8 Test: Reset With and Without Ethernet Connected

Verify that the target will properly run through a reset with and without the Ethernet cable connected. Sometimes the drivers are accidentally written in such a way that the port connectivity is required; otherwise, the driver sits in a loop waiting for link up. This test just confirms that not to be the case.

11.14.9 Test: Run "icmp echo 192.168.1.100"

To verify basic network connectivity, try to "ping" an external device. If you know the IP address of device on your network, then issue the "icmp echo" (i.e. ping) to that address. Assuming the target address is alive and well, then the icmp echo command will immediately respond with "192.168.1.100 is alive"; otherwise it will retry a few times prior to giving up. Note that a failure here may be due to an unresponsive target, so make sure you are using a valid, IP address.

11.14.10 Test: Run moncmd

The monitor has the ability to process incoming commands over UDP. This, as described in section 2.6, is done with the host-based tool called moncmd. To test this functionality simply run issue a remote command to your target's IP address.

11.14.11 Test: Verify LOOPS_PER_SECOND Setting

The monitor's default mode of timing is with a simple loop count that is set in config.h (LOOPS_PER_SECOND). Issue the command "sleep 1" and verify that the delay is close to a second. To recalibrate the value to be used in LOOPS_PER_SECOND, use the "-c" (configure) option of sleep. Refer to the sleep manpage for details.

11.14.12 Test: Using MonStack?

The monitor code assumes that the stack is within the array MonStack[] declared in start.c and the size of MonStack[] is defined by the declaration of MONSTACKSIZE in config.h. It's easy to accidentally use some other allocated block of memory for the stack, so verify in reset.S that MonStack is being used.

11.15 Wrap Up

So, while it may look like a lot of detail, the basic port of uMon to any target is fairly simple. A polled UART driver is all it takes to get started. Following are a few suggestions that may prove helpful during your porting process…

- It is important to **take little steps** when doing a port. The config.h INCLUDE_XXX macros allow you to do this. Don't try to build the whole platform in one shot. One step at a time will get you there a lot faster!!!
- The list of configurable items may seem a bit overwhelming at first, so start a new port with what is defined in the template config.h file, then after stablizing the port, you can then start to tune these parameters as needed.
- **Don't enable cache initially**. Just leave it off until you have the port complete and you build up some confidence in what you have.

- **Make sure optimization is turned off initially**. Eventually, you can enable it and things should be made to work with optimization; however, eliminate that complexity when starting a new port.
- In all of the driver polling loops insert some type of timeout. If the timeout occurs, depending on where it occurs, you can do a few different things: if the timeout occurs in a UART-related poll, then blink an LED, or do SOMETHING to let you know where the code is. If the timeout occurs somewhere after the uart interface is known to be up and running, then call some simple error message that will print out the serial port.
- Always start a port with FORCE_BSS_INIT enabled. The guarantees that the bss space is properly initialized at startup.
- When writing warmstart code (usually in reset.s), be aware that it may be called by an actual reset (or power cycle) or it may be called as a result of an exception. If called by the monitor's exception handler, then it is important to flush data cache prior to disabling it (at startup). This is important because the exception hander loads a table with the current context of the CPU (registers), so if that data space is in cacheable space, and it isn't flushed, it will be lost; hence, not retrievable by the monitor for debugging the cause of the exception. Ideally, the caches should only be disabled/invalidated in coldstart (prior to warmstart).

Chapter 12 Miscellaneous Application Notes

This chapter is dedicated to short discussions about various topics within uMon that do not necessarily fit well within any of the previous chapters. The subject matter is random, strictly based on the need for the discussion. It also includes a growing list of "How Do I…" topics that originated from various questions I've received over the years regarding different features in uMon.

12.1 Runtime Reconfiguration of TFS

At the time of installation, a target running MicroMonitor has some block of flash allocated to TFS. Usually, the default configuration is for TFS to just overlay all of flash space. This is fine for systems that only use TFS as their flash file system (FFS); however, for systems that may want to boot with MicroMonitor, then run their OS with some other FFS overlaying a portion of the flash, it's impractical. The "cfg" sub command within TFS allows the user to modify the amount of flash space that is owned by TFS on a given target. This allows a target system to be initially configured (at uMon build time) with some TFS allocation, then when in the field, adjustments can be made without the need to rebuild uMon. The following steps modify TFS's flash configuration. Obviously the actual values used (and displayed) will depend on your target:

Determine which sectors in the flash are owned by TFS. This is done with the "flash info" command. As shown below, this command dumps information about each of the sectors in the target's flash area. Notice the second column. If an asterisk is present in that column, this indicates that the sector is occupied by TFS for either storage or spare sector usage.

```
uMON>flash info
Current flash bank: 0
Device = AMD-29DL160DB
 Base addr   : 0xffe00000
 Sectors     : 35
 Bank width  : 2
 Flash ops   : relocated

 Sctr TFS?   Begin       End         Size    SWProt?  Erased?
   0   *   0xffe00000  0xffe03fff  0x004000    no        no
   1   *   0xffe04000  0xffe05fff  0x002000    no        no
   2   *   0xffe06000  0xffe07fff  0x002000    no        no
   3   *   0xffe08000  0xffe0ffff  0x008000    no        no
   4   *   0xffe10000  0xffe1ffff  0x010000    no        no
   5   *   0xffe20000  0xffe2ffff  0x010000    no        no
   6   *   0xffe30000  0xffe3ffff  0x010000    no        no
   7   *   0xffe40000  0xffe4ffff  0x010000    no        no
   8   *   0xffe50000  0xffe5ffff  0x010000    no        no
   9   *   0xffe60000  0xffe6ffff  0x010000    no        no
  10   *   0xffe70000  0xffe7ffff  0x010000    no        no
  11   *   0xffe80000  0xffe8ffff  0x010000    no        no
  12   *   0xffe90000  0xffe9ffff  0x010000    no        no
  13   *   0xffea0000  0xffeaffff  0x010000    no        no
  14   *   0xffeb0000  0xffebffff  0x010000    no        no
  15   *   0xffec0000  0xffecffff  0x010000    no        no
  16   *   0xffed0000  0xffedffff  0x010000    no       yes
  17   *   0xffee0000  0xffeeffff  0x010000    no       yes
  18   *   0xffef0000  0xffefffff  0x010000    no       yes
  19   *   0xfff00000  0xfff0ffff  0x010000    no       yes
  20   *   0xfff10000  0xfff1ffff  0x010000    no       yes
  21   *   0xfff20000  0xfff2ffff  0x010000    no       yes
  22   *   0xfff30000  0xfff3ffff  0x010000    no       yes
```

```
  23   *   0xfff40000  0xfff4ffff  0x010000    no       yes
  24   *   0xfff50000  0xfff5ffff  0x010000    no       yes
  25   *   0xfff60000  0xfff6ffff  0x010000    no       yes
  26   *   0xfff70000  0xfff7ffff  0x010000    no       yes
  27       0xfff80000  0xfff8ffff  0x010000   yes        no
  28       0xfff90000  0xfff9ffff  0x010000   yes        no
  29       0xfffa0000  0xfffaffff  0x010000   yes        no
  30       0xfffb0000  0xfffbffff  0x010000   yes        no
  31       0xfffc0000  0xfffcffff  0x010000   yes        no
  32       0xfffd0000  0xfffdffff  0x010000   yes        no
  33       0xfffe0000  0xfffeffff  0x010000   yes        no
  34       0xffff0000  0xffffffff  0x010000   yes        no
uMON>
```

Using the output of the above command as an example, we can see that TFS spans from `0xffe00000` to `0xfff7ffff`. Now, assume we want to release the last 7 sectors (20-26) for use by some other FFS or just for general flash storage outside the domain of TFS. There are a few issues that must be dealt with prior to being able to reconfigure TFS:

- Make sure the flash space that you are going to release from TFS is erased. In this case, that would be the area from `0xfff10000` through `0xfff7ffff`. This may require that "tfs clean" be run so that the area is erased, or it may first require that you remove some files (or all files using "tfs init") just to erase this area.
- The area of flash that "tfs cfg" actually modifies is not part of TFS, its part of uMon's executable binary image. That flash area is probably protected, so it will require that you first "unprotect" and possibly "unlock" the flash[82]. The easiest way to do this is to just unlock/unprotect all of flash with the command sequence: "`flash unlock all; flash unprot all`".

Now we're ready to reconfigure TFS. To do this we use the "tfs cfg" command. The usage of this command is:

```
tfs cfg {start_addr} {end_addr}
```

so, in our case, we issue the command:

```
tfs cfg 0xffe00000 0xfffffff
```

to change the control structures within uMon that tell it what portion of flash is allocated to TFS. Note that the "end_addr" value is the end address of the space used for TFS file storage, so it is assumed that the next location is the start of the spare sector; hence, one additional sector of space is still going to be used by TFS.

The end result is that TFS now occupies a smaller area of the flash, thus allowing some portion of the flash to be used by something other than TFS. Notice the new output of "flash info" below. There is no longer an asterisk next to those 7 sectors (20-26); hence, TFS does not own them…

```
uMON>flash info
Current flash bank: 0
Device = AMD-29DL160DB
 Base addr   : 0xffe00000
 Sectors     : 35
 Bank width  : 2
 Flash ops   : relocated
```

[82] Even if this space isn't locked or sw-protected, these command s can be executed, so it's best to just run them in all cases.

```
 Sctr TFS?   Begin       End         Size    SWProt?  Erased?
   0   *   0xffe00000  0xffe03fff  0x004000    no        no
   1   *   0xffe04000  0xffe05fff  0x002000    no        no
   2   *   0xffe06000  0xffe07fff  0x002000    no        no
   3   *   0xffe08000  0xffe0ffff  0x008000    no       yes
   4   *   0xffe10000  0xffe1ffff  0x010000    no       yes
   5   *   0xffe20000  0xffe2ffff  0x010000    no       yes
   6   *   0xffe30000  0xffe3ffff  0x010000    no       yes
   7   *   0xffe40000  0xffe4ffff  0x010000    no       yes
   8   *   0xffe50000  0xffe5ffff  0x010000    no       yes
   9   *   0xffe60000  0xffe6ffff  0x010000    no       yes
  10   *   0xffe70000  0xffe7ffff  0x010000    no       yes
  11   *   0xffe80000  0xffe8ffff  0x010000    no       yes
  12   *   0xffe90000  0xffe9ffff  0x010000    no       yes
  13   *   0xffea0000  0xffeaffff  0x010000    no       yes
  14   *   0xffeb0000  0xffebffff  0x010000    no       yes
  15   *   0xffec0000  0xffecffff  0x010000    no       yes
  16   *   0xffed0000  0xffedffff  0x010000    no       yes
  17   *   0xffee0000  0xffeeffff  0x010000    no       yes
  18   *   0xffef0000  0xffefffff  0x010000    no       yes
  19   *   0xfff00000  0xfff0ffff  0x010000    no       yes
  20       0xfff10000  0xfff1ffff  0x010000    no       yes
  21       0xfff20000  0xfff2ffff  0x010000    no       yes
  22       0xfff30000  0xfff3ffff  0x010000    no       yes
  23       0xfff40000  0xfff4ffff  0x010000    no       yes
  24       0xfff50000  0xfff5ffff  0x010000    no       yes
  25       0xfff60000  0xfff6ffff  0x010000    no       yes
  26       0xfff70000  0xfff7ffff  0x010000    no       yes
  27       0xfff80000  0xfff8ffff  0x010000    no        no
  28       0xfff90000  0xfff9ffff  0x010000    no        no
  29       0xfffa0000  0xfffaffff  0x010000    no        no
  30       0xfffb0000  0xfffbffff  0x010000    no        no
  31       0xfffc0000  0xfffcffff  0x010000    no        no
  32       0xfffd0000  0xfffdffff  0x010000    no        no
  33       0xfffe0000  0xfffeffff  0x010000    no        no
  34       0xffff0000  0xffffffff  0x010000    no        no
uMON>
```

That's all there is to it. The space from `0xfff00000` thru `0xfff7ffff` is now available as raw flash, unoccupied or touched by TFS. Note the output of "tfs stat" at this point…

```
uMON> tfs stat
TFS Memory Usage...
      name    start       end       spare     spsize  scnt type
  //FLASH/: 0xffe00000|0xffefffff|0xfff00000|0x010000|  19|0x200000

 Total memory: 1114112 bytes (used=95656, avail=1018456 (1018300 for data)).
 Per-device overhead: 65628 bytes (defrag-state=92 spare-sector=65536).
 File data space: 28780 bytes (live=28780, dead=0).
 File overhead space: 1248 bytes (live=1248, dead=0).
 File count: 8 (live=8, dead=0).
 Defrag will release 0 bytes

TFS Hdr size: 92
Total files currently opened: 0
```

```
uMON>
```

The output shows that the end and spare location are now adjusted to the new address range. The user needs to be aware of the possibility of specifying an incorrect address, plus if the adjustment to the TFS space conflicts with files that are currently in TFS storage area, it may be necessary to defrag TFS or remove some files.

12.1.1 A Few Final Notes on "tfs cfg"...

1. If this is not the first re-configuration of TFS space for this target, then the "tfs cfg {start} {end}" command must be preceded by "tfs cfg restore" so that uMon removes the previously written configuration first. This step simply restores the TFS configuration to what it was built with. **During this step, the sector that contains that data structure within uMon's executable image is erased, modified and re-written, so don't interrupt that step[83] (a warning will be printed to the console).** It usually takes 3-5 seconds to complete and includes an automatic restart of uMon[84]. Once this is done, then "tfs cfg {start} {end}" can be issued.
2. uMon's ability to re-write the TFS control structure is available as of release 1.8; however, it is still dependent on some implementation in the port-specific code. To know if your target is capable of re-writing the TFS control structure, run the "help –i" command. If the output includes a line that identifies the address of the control structure (for example: `AltTFSdevtbl: 0x00000050`), it is enabled.

12.2 Creating a RAM Based TFS Storage Area

There are two fundamental ways to configure TFS to overlay onto RAM: statically at build time or dynamically at run time. These two different mechanisms can be used independently or simultaneously. The main difference between the two is that one is hard-configured (it is seen by uMon immediately out of reset) and one is dynamically configured (it is created at runtime by a command in uMon). The static mechanism has been around for quite a while, the runtime method is part of uMon as of uMon1.0.

12.2.1 Static TFS RAM Overlay

This configuration is set up essentially the same way a flash-based TFS partition is set up. For each partition there is a tfsdev structure entry in the tfsdev structure table found in the port-specific file tfsdev.h. The only difference here is that the device type is different. For flash, the devinfo field typically is set to TFS_DEVTYPE_FLASH. For ram, the device type will be TFS_DEVTYPE_RAM or TFS_DEVTYPE_NVRAM. The main difference here is that NVRAM will not be automatically cleared unless the added TFS_DEFINFO_AUTOINIT flag is also set in this field. All other entries in this structure have the same meaning for ram as they have for flash.

12.2.2 Dynamic TFS RAM Overlay

As of uMon1.0, TFS supports a dynamically configurable RAM based TFS partition. The new tfs sub-command "ramdev" allows the user to allocate a block of RAM for TFS storage at any time. This allows a developer to configure some RAM space to be used temporarily for file storage, and if it turns out to be part of the application, then the "tfs ramdev" command can simply be added to the monrc file. The primary use of this feature is to allow a developer to hack away at TFS without really touching any flash. Typical use would be during development of an application destined for TFS space; however, while developing new versions of the code, the file can be downloaded into RAM-based TFS just to reduce the wear and tear on the flash (plus it reduces the defragmentation overhead).

Following is a usage example for the tfs subcommand "ramdev". We'll assume for this example that our board's ram space extends from 0 thru 32Mg (`0x00000000 – 0x02000000`), and we are going to allocate the last 2Mg (0x200000) to TFS for RAM based storage. The following command would be used:

[83] Its VERY important that this step be un-interrupted because if not allowed to complete successfully, the boot flash is likely to be corrupted. It should take between 3-6 seconds to complete.

[84] Depending on the implemention of the soft reset for any given port, this may not automatically reset; hence, if after 10-15 seconds the board has not reset, a manual reset may be needed.

```
tfs ramdev TMP 0x1e00000 0x200000
```

to allocate a new TFS device named “TMP”. The output of “tfs stat” shows the new device:

```
uMON>tfs -d //TMP/ stat
TFS Memory Usage...
      name    start       end       spare     spsize  scnt type
    //TMP/: 0x01e00000|0x01ffffff|  - NA -  | - NA - | NA |0x100000
```

Then files can be added as “//TMP/filename” and TFS will automatically be placed within that block of RAM space just as if it was flash (except that it is lost on power up). For example, assuming you have a file called “monrc” in your normal TFS space, you can now issue the command “tfs cp monrc //RAM/monrc_copy” and it will show up in TFS as follows…

```
uMON>tfs ls
 Name                       Size   Location   Flags  Info
 //TMP/                             (dir)
 monrc                       105  0x1a00005c  e

Total: 2 items listed (105 bytes).
uMON>tfs ls *
 Name                       Size   Location   Flags  Info
 //TMP/monrc_copy            105  0x01e0005c  e
 monrc                       105  0x1a00005c  e

Total: 2 items listed (210 bytes).
uMON>
```

Notice that on the first “tfs ls” output, the location is shown by TFS as (dir). This is TFS trying to be a little intelligent with your files. All its doing is showing you that there is at least one file with the //TMP/ prefix; don’t get crazy and assume that TFS supports a directory hierarchy! The second listing above shows the output of “tfs ls *” and in that case, the actual file is shown. Had there been more than one file, then all would be seen in the second listing.

12.2.3 Flashless TFS

One final topic worth mentioning here is that while TFS inherently wants to be hooked up to flash, a system can be configured to only have a RAM-based TFS partition(s). This can be done using either (or both) of the above modes. If a statically allocated RAM partition is used, then simply make that entry in the tfsdev.h table. If there is to be no statically allocated RAM partition, but there may be an occasional need to treat some RAM as file-storage space, then the tfsdev.h table can be configured with only an empty terminator entry…

```
struct tfsdev tfsdevtbl[] = {
    { 0, TFSEOT,0,0,0,0,0 }
};
#define TFSDEVTOT((sizeof(tfsdevtbl))/sizeof(struct tfsdev)))
```

then the “tfs ramdev” command can be used at runtime to create the partition as needed. Either way works; however note that since there is no permanent file storage there will be no “automatic-on-bootup” capability simply because there won’t be any scripts in TFS at startup.

12.3 Voluntarily Updating Your Monitor Image

Assuming the flash drivers are written correctly, uMon supports the ability to install a new monitor over top of the one that is currently running. This convenience is both good and bad. Good because it allows you to

update without the need of any kind of JTAG-like attachment, bad because **if you get it wrong, you destroy the boot image of your target system** and then you do need external hardware to restore the boot image. The point here is that if you aren't sure about what you're doing, and you do not have the capability to restore your boot image with some external tools then don't just do this to "experiment" with your target. There are two ways to do this update. One uses the serial port and the other uses Ethernet…

12.3.1 xmodem –B:

The Xmodem command allows the user to download images to memory. The –B option to Xmodem tells it to do all the steps necessary to transfer the downloaded image into the portion of flash that contains the boot monitor. So, after building a new raw binary image of the monitor, the image is simply transferred to the target using xmodem –B. Upon completion of the transfer, and prior to doing the actual boot-image update, xmodem –B will query the user one last time…

```
Reprogramming boot @ 0xBBBB from 0xAAAA, SSSS bytes.
OK?
```

Translated… 'SSSS' bytes will be copied from RAM space beginning at 0xAAAA to boot flash space beginning at 'BBBB. The value of '0xAAAA' is usually the content of $APPRAMBASE, the value of '0xBBBB' is the content of $BOOTROMBASE, and 'SSSS' is the size of the file just transferred. Respond to the query with 'y' to continue ONLY if this information is correct; else abort with any other character.

If your flash device supports sector locking, and your monitor was built to support it[85], then be aware that the xmodem –B command automatically unlocks all flash sectors, so if sectors were locked prior to this command, they must be manually re-locked after completion of this command. This automatic unlock is relatively new (Aug 2004), so it may not be part of your running monitor. In that case, prior to issuing the "xmodem –B" command, run "flash opw" followed by "flash unlock RNGE" where 'RNGE' is the sector range that you want to unlock. Depending on the flash device, the RNGE value may be ignored because the unlock may not be per-sector, it may apply to the whole device.

12.3.2 newmon:

The tool "newmon" (see section 17.17 below) can be used to burn a new monitor image. Depending on the platform on which it is run, newmon is either a script (Unix) or a single executable (windows). In either case, newmon does two things (similar to xmodem –B, just faster because it uses Ethernet for the file transfer)..

Download the binary image (via Ethernet).
Transfer the binary image to the boot space.

Both versions (Unix script & windows .exe) take two arguments: the IP address of the target and the file containing the image destined for the boot flash. The windows version includes some additional options that generally are not needed. For more information on the newmon.exe tool, refer to 17.17. If you're on Unix, then just read through the newmon script.

Similar to the xmodem –B discussion above, your target may support lockable sectors. If it does, then prior to running newmon, you should unlock the sectors that are used by MicroMonitor so that the flash operations are successful.

12.4 Using an External Debugger (JTAG or similar)

MicroMonitor provides some basic debug/diagnostic tools; however, it doesn't provide single stepping and breakpoint capabilities. For real tough problems, nothing replaces a good external hardware based debugger connected to the CPU's JTAG or BDM port. That's where an external debugger comes in handy. There are a few different choices available, the two that immediately come to mind are from Abatron and Macraigor. For the sake of this discussion, I will be referring to the "BDI2000", the JTAG-based debugger from Abatron; however, note that this discussion applies to any uMon situation in which an external debugger is used.

With a debugger attached, things are a bit different. In most cases, the debugger connects to the target, immediately takes control of the CPU and does not allow the boot code (uMon) to run. The CPU is controlled

[85] The output of the "help flash" command will have a "lock/unlock/lockdwn" sub-command listed if flash locking is enabled in your monitor.

by the JTAG interface so each instruction fetch can be examined by the developer. This is good stuff; however, with it comes the need to be aware of what else might be different. The target doesn't boot with MicroMonitor, it doesn't even boot. Upon power up of the debugger and target, the debugger takes control; essentially halting any instruction fetches until told to do something. In a normal uMon-based application, the application is launched by the monitor out of TFS. The application hooks up to uMon through the monConnect() facility and is able to legally assume that the monitor is up and running. This is not the case with BDI2000 based debugging; hence, a few things have to be considered. Commands within the bdi2000.cnf file are used to do some very basic initialization of the target, but that's about it. This usually includes establishing valid access to the DRAM and flash. You need to realize that the BDI2000 is helping to debug the application, not the monitor itself; hence, it is unaware of the monitor's underlying presence. Immediately, the start() function of the application must change…

No stack pointer. When launched from MicroMonitor, the application can assume that it is at least initially running off of the monitor's stack. This is because the entry point of the application is branched to from MicroMonitor just like any other function; hence, the stack of MicroMonitor is used by the early stages of the start() function of the application. The creation of a stack has a slightly different meaning now. Instead of allocating a stack so that it can isolate itself from the monitor's stack, the creation of the stack is done now just so that there is a stack pointed to by the stack pointer; hence, it needs to be done at the top of the start() function, possibly in assembler. It must be set to point to some block of memory that is known to be useable as stack space (refer to section 9.4 for details on doing this). Note that since the BDI2000 will download the application into RAM somewhere on the target, that download must be able to assume that the RAM space that the application is destined for has been initialized. This initialization must be done in the bdi2000.cnf file.

No Initialized MicroMonitor. Since the target's bootup was interrupted by the BDI2000, MicroMonitor never ran; hence, if it is going to be used by the application, then it's internal state needs to be initialized. In a normal standalone system, the monitor boots up and initializes its own state, then turns over control to the application. The application can then hook up to the monitor and use some of its facilities. Now with the BDI2000 having halted the CPU prior to MicroMonitor booting up, the application is started but MicroMonitor isn't even initialized. This is dealt with immediately after the monConnect() call, there is a need to call the monitor API function mon_warmstart(). This API is provided so that the application can initialize the monitor; hence, establishing the monitor's own state prior to having the application use any other monitor hooks. Once this call is completed, the application can assume that the monitor's state is initialized and ready for use.

Ideally, there should be some way in the startup code to detect whether or not the application is being launched by the monitor or by the BDI2000. This would allow it to automatically figure out whether or not it is necessary to make the call to mon_warmstart(). This is something that is likely to be target specific; hence, for this example we will simply use a #define…

```
01: #include "monlib.h"
02:
03: #define USING_BDI2000   1
04:
05: /* Defined in the memory map file...
06:  */
07: extern char _bss_start, _bss_end;
08:
09: extern int main(int argc, char *argv[]);
10: unsigned long AppStack[1024];
11:
10:
11: void
12: Cstart(void)
13: {
14:     char    **argv;
15:     int     argc, ret;
16:     volatile int i, j;
19:     volatile register char     *ramstart asm ("a0");
20:
21:     /* Initialize application-owned BSS space.
22:      * If this application is launched by TFS, then TFS does
```

```
23:      * it automatically, however since MicroMonitor provides
24:      * other alternatives for launching an application, we
25:      * clear bss here anyway (just in case TFS is not launching
26:      * the app)...
27:      */
28:     ramstart = &_bss_start;
29:     while(ramstart < &_bss_end)
30:         *ramstart++ = 0;
31718:
3219:     /* Connect the application to the monitor.  This must be done
330:      * prior to the application making any other attempts to use
341:      * the "mon_" functions provided by the monitor.
352:      */
363:     monConnect((int(*)())(*(unsigned long *)0xff800008),(void *)0,(void *)0);
374:
385: #if USING_BDI2000
396:     mon_warmstart(WARMSTART_ALL);
407: #endif
418:
4229:     /* Extract argc/argv from structure and call main(): */
430:     mon_getargv(&argc,&argv);
441:
452:     /* Call main, then return to monitor. */
463:     ret = main(argc,argv);
474:
485:     /* Since we established a stack frame, we can't just return to
496:      * the monitor.  We have to exit...
507:      */
518:     mon_appexit(ret);
5239: }
530:
```

• Listing 54 : The Cstart() Function for BDI2000

Three primary differences between this startup and previous (non-debugger) startup files…

- The re-establishment of the stack frame is now a requirement, and actually not a bad thing with or without the external debugger.
- The initialization of BSS space is required since the application is not being launched out of TFS (which automatically clears bss space prior to jumping into the application's entrypoint).
- The call to mon_warmstart() immediately after monConnect(). This is the hook that allows the application to actually start the monitor. It is essential when using the BDI 2000 to debug an application that assumes it can take advantage of the MicroMonitor facilities.

12.5 How Do I Get the Size and/or Location of a File in TFS?

There are two different contexts under which one might want to retrieve this information: in a script or in an application program.

12.5.1 Size and Location From a Script:

Given the following set of files, with "script" being the file that will provide the demonstration…

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 app                      71428  0xfff0f3cc  E
 monrc                      114  0xffe0005c  e
 script                     105  0xfff21aac  e
 symtbl                    3582  0xfff20b2c
 wakeup                      44  0xfff21a1c  Be

Total: 5 items listed (75273 bytes).
```

```
uMON>tfs cat script
tfs size app APPSIZE
tfs base app APPBASE
echo The file 'app' starts at $APPBASE, and is $APPSIZE bytes.
uMON>script
The file 'app' starts at 0xfff0f3cc, and is 71428 bytes.
uMON>
```

The script called “script” (shown above) retrieves the size and base address of the file “app” using the “tfs” command. Execution of the script shows the output that matches the above listing.

12.5.2 Size and Location From a Program:

The size and base of a file can be used in a program if the program needs to access the file through direct memory. The following snippet of code demonstrates this…

```
#include “tfs.h”

void
some_function(void)
{…
        TFILE *tfp;
        long size;
        char *base;

        tfp = mon_tfsstat(“filename”);
        size = TFS_SIZE(tfp);
        base = TFS_BASE(tfp);

        mon_printf(“The file \”app\” starts at 0x%lx, and is %d bytes.\n”,
                (long)base, size);
}
```

12.6 How Do I Intercept uMon Command Output in my Application?

uMon’s API allows a user to hook to uMon’s command line interface from within an application. This means that an application’s command line interface can immediately include the full set of commands that are provided by uMon. Refer to the mon_docommand() API call (section 16.10) for more information on this. However, what if you want to have the equivalent of uMon’s MONCMD server in my application? Now you need the ability to issue the command (using mon_docommand()), but you also need the ability to catch all the output of that command so that it can be passed back to the UDP client that made the request. The following example code assumes that your application has the standard UDP sendto() and recvfrom() functions available.

The console I/O functions in uMon can be redirected to a function of your choice. Hence, you can temporarily have uMon's putchar() function be a function that is in your application, and as a result, you can pump each line of uMon's response out your UDP socket. Here's some pseudo code...

```
static char mybuf[128];
static *myptr;
int myputchar(char c);
void send_mybuf_to_client(void);

void
my_moncmd_srvr(void)
{
```

```
        ...

        /* Wait for packet from client and assume it to be a uMon command...
         */
        recvfrom(blah-blah-blah, cmdline);


        /* Temporarily re-assign uMon's putchar function to be an
         * application-specific putchar function...
         */
        mon_com(CHARFUNC_PUTCHAR,myputchar,0,0);


        /* Issue command to monitor.
         * While this is running, the myputchar() function will be
         * called...
         */
        mon_docommand(cmdline,0);

        /* Restore the original functionality back into uMon...
         */
        mon_com(CHARFUNC_PUTCHAR,0,0,0);

        ...
}



/* myputchar():
 * Accumulate input until a full line is received, the pass that line
 * back to the client...
 */
int
myputchar(char c)
{
        if ((myptr == 0) || (myptr >= mybuf + sizeof(mybuf) - 2))
                myptr = mybuf;
        *myptr++ = c;
        if (c == '\n') {
                *myptr = 0;
                send_mybuf_to_client();
                myptr = mybuf;
        }
}

/* send_mybuf_to_client():
 * This function is called when myputchar() has accumulated a full line of text
 * from the output of the uMon command called via mon_docommand() in my_moncmd_srvr().
 */
void
send_mybuf_to_client(void)
{
        ...
        sendto(blah-blah,mybuf,strlen(mybuf));
        ...
```

```
}
```

Refer to section 8.2 of the user's manual for more details.

12.7 How Do I Attach the Date to a File in TFS?

TFS can be configured to support automatic insertion of time/date into a created file's header; however, that requires that the target board have the ability to maintain time, and also requires a hook to be part of the uMon port so that TFS can retrieve the time/date from the source. Bottom line is that in the majority of cases, this isn't practical because the board doesn't keep track of time. So, the next best thing is to be able to attach some idea of time to the file in the "info" field of the file transferred to TFS. In all cases, note that the "info" field can be used to store whatever text (or none at all) is useful to describe the content of the file. This section just demonstrates a few different ways that this field can be used to store time/date info using tftp…

12.7.1 Attach the Host's Time-of-Last-Modification to a TFS File

Assume you have a file on your host machine…

```
unixhost:ls -l
total 3
-rwxrwxrwx    1 Ed       mkpasswd      944 Sep 23 16:18 Entries
-rwxrwxrwx    1 Ed       mkpasswd       26 Sep 23 16:18 Repository
-rwxrwxrwx    1 Ed       mkpasswd       43 Sep 23 16:18 Root
unixhost:
```

Now assume you want to transfer the "Entries" file to your target and you want the info field to reflect the same modification date as is on the host (Sep, 23 16:18). The following lines can be used in a shell script to create this file and info field…

```
tfsname=`ls -l Entries | awk '{printf "Entries,,%s_%s_%s\n",$6,$7,$8}'`
ttftp 1.2.3.4 put Entries $tfsname
```

The shell variable "tfsname" (on the host side) is loaded with the string "`Entries,,Sep_23_16:18`" , then that name is used as the destination name by ttftp when the file is transferred to the target. The result is that a "tfs ls" command will now show the modification time of the file as it was on the host at the time of the transfer…

```
uMON>tfs ls
 Name                        Size   Location   Flags  Info
 Entries                      944  0x107dbb1c         Sep_23_16:18
 dhcpwait                     125  0x1047b66c  e
 linux/                           (dir)
 mine                      101592  0x1008005c  Ec
 monrc                        473  0x107db8dc  e      envsetup

Total: 5 items listed (103134 bytes).
uMON>
```

12.7.2 Attach the Host's Current Time to a TFS File

As an alternative to 12.7.1 above, perhaps the desired entry in the info field should be the time of the actual transfer of the file to TFS (this is the value that would be automatically added to TFS's file header if the board maintained time-of-day). The difference lies in the "tfsname=" line above…

```
tfsname=` date +Entries,,%b%d_%Y_%H:%M`
ttftp 1.2.3.4 put Entries $tfsname
```

The result in TFS is similar…

```
uMON>tfs ls
 Name                        Size   Location   Flags  Info
 Entries                      944  0x107dbb1c         Sep23_2006_16:18
 dhcpwait                     125  0x1047b66c  e
 linux/                           (dir)
 mine                      101592  0x1008005c  Ec
 monrc                        473  0x107db8dc  e      envsetup

Total: 5 items listed (103134 bytes).
uMON>
```

Chapter 13 Topics Specifc to Booting Embedded Linux

The MicroMonitor platform is used as a startup environment for many different embedded system applications. Under the context of this discussion, there are three main types of applications...

- Those that run single-threaded, without any need for an embedded operating system.
- Those that run multi-threaded, using an RTOS that runs with a flat, basically unprotected memory space.
- Those that run multi-processed, using an RTOS that provides memory protection between processes.

For the sake of this discussion, the first two types can be bunched together, leaving us with those applications that run with per-process memory protection and those that do not. This chapter is intended for those that run with per-process memory protection (i.e. embedded Linux). There are some distinct differences and limitations that apply to this type of environment, and the intent here is to bring these limitations forward.

MicroMonitor was booting embedded systems years before embedded Linux and other "process-oriented" operating systems were practical for embedded projects. As a result, there are a lot of application-run-time features in uMon that are not useful when running with an operating system that uses the MMU for per-process protection. This is a limit imposed by the operating system, not the bootloader, and it is true regardless of the bootloader used. That being said, it should not imply in any way that uMon isn't applicable for booting those systems, it simply means that those systems can't access as much of uMon's runtime facilities because of memory protection.

This chapter discusses a few different topics that are likely to be of interest to those considering (or already using) uMon to boot an embedded operating system such as Linux. Hopefully at the end of this chapter it will be clear that there's really "nothing to be afraid of" when booting embedded Linux. It's a much more complicated beast than most other RTOSes (but also provides a heck of a lot of useful features); hence, it requires some unique consideration for booting. The complexity of uMon is not increased when booting Linux, there's just a few additional commands and features that you may want to consider enabling in uMon if you plan to support booting embedded linux with your uMon port. This chapter will dicsuss…

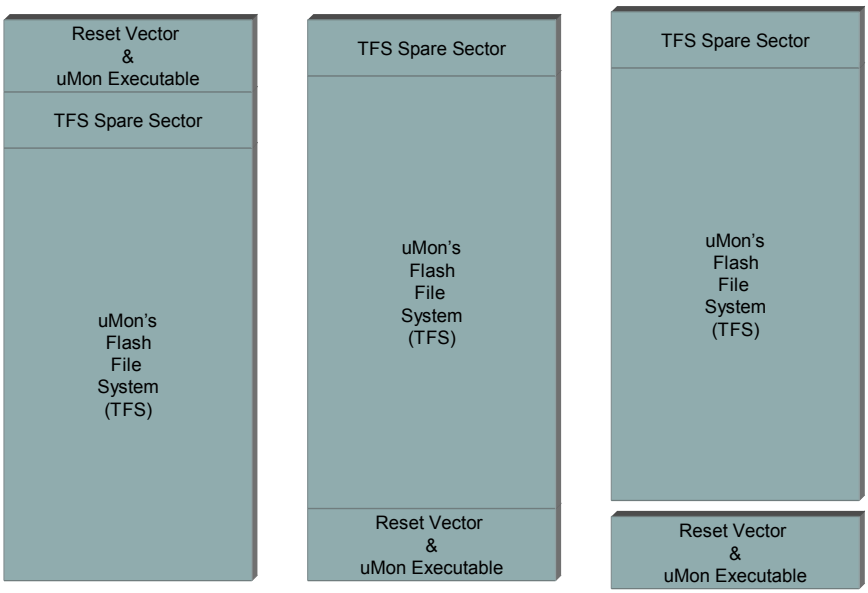
- configuring flash so that TFS and other embedded flash file systems can co-exist even in the same flash device
- using scripts to allow linux to boot in one of several different ways depending on the need
- using the 'lboot' command to start up a kernel
- using the 'ldatags' command to configure a kernel
- using the 'jffs2' command to help decide how to boot
- accessing TFS files at the linux prompt
- etc…

13.1 Configuring Flash with uMon and Embedded Linux

The purpose of this section is to discuss various ways that flash can be configured in a MicroMonitor (i.e. uMon) based embedded Linux application.

13.1.1 A Simple, Non-MMU Based Application Configuration

A simple uMon-based embedded target has a flash map that is broken up into two main sections: the uMon executable and TFS. Then, assuming TFS is configured with power-safe defragmentation, the spare sector is the top-most sector of the flash area allocated to TFS. Referring to Figure 10, the relative position of the uMon executable and TFS is arbitrary. The executable may reside above TFS, below TFS or even on a separate device that is in non-contiguous memory space.



• **Figure 10: Basic uMon Flash Memory Map**

Certainly the TFS configuration can get more elaborate. There may be reason to spread TFS across different non-contiguous devices, or even have multiple TFS partitions within a single device. It can also be configured for use in RAM either statically at build time or dynamically at runtime.

When running with a non-MMU based operating system, TFS may be, and usually is, the only flash file system needed for both bootup and application runtime. At bootup, uMon accesses TFS to retrieve the files that make up the boot (usually some script and an .elf file to be loaded into RAM), then once the application is up and running it too has access to uMon's API; hence, it can read/write files from/to TFS just as easily as uMon can. The uMon API provides the application with hooks to functions that are in uMon's executable space, that is, outside the mapped space known by the application. This is ok because there's no MMU protection; hence all memory is accessible. It works just fine.

13.1.2 An MMU Based Application (i.e. Linux) Configuration

Embedded Linux presents two additional issues that need to be dealt with (or at least understood) when using uMon as the boot loader. Embedded Linux runs with the MMU enabled in such a way that it is impossible to access the uMon API from user space. In addition, even if the API was accessible, TFS is not a native file system to Linux. These are problems that any boot loader has to face; however, because one of uMon's assets is TFS, we need to consider ways to boot Linux and still take advantage of what the TFS/uMon combination has to offer.

As is typically the case for embedded systems, the solution depends on the problem; hence, there are several different configurations, each of which have their own applicability depending on the system requirements. So, lets walk through a few different configurations…

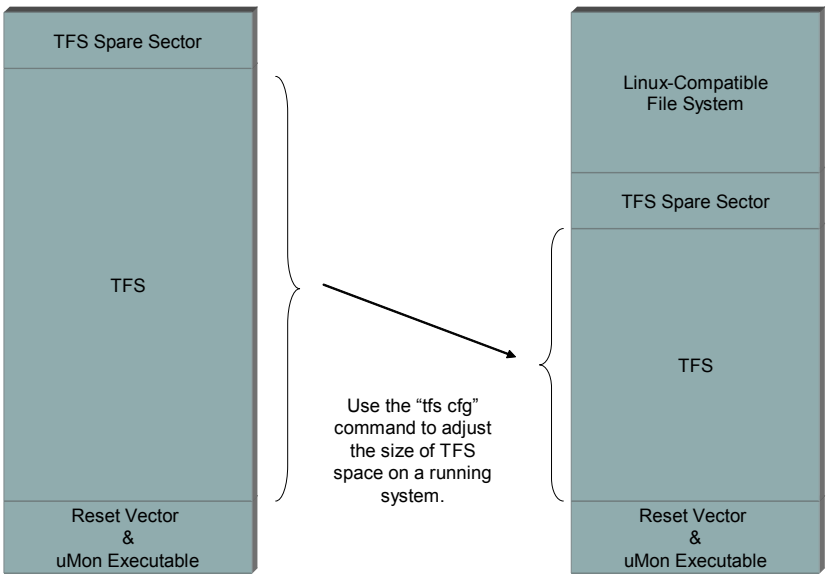
Good: TFS The Only FS

In this case the flash is mapped similar to that of Figure 10, with both the kernel and FS images in TFS as files. Then a simple, ASCII uMon startup script is used to decompress the files into their respective locations

and jump into the entrypoint of the kernel[86]. In this simplest of cases, all on-board file system space accessible by Linux is RAM; hence, no permanent storage for use by the application. While this is somewhat limited, it may be all that is needed, and to put a positive spin on it, the running linux application can't corrupt anything used at boot up; hence, the system is quite secure from any rogue Linux application.

Better: TFS for Boot, Linux-FS for Runtime

Here we have a slightly more complicated flash footprint. We still need a section of memory for the boot monitor executable and for TFS storage space, but now the TFS storage space is reduced and some other Linux-compatible FFS is overlayed on a portion of the flash. Referring to the map of Figure 11, all images are in a single flash device containing the uMon executable, TFS and a Linux-compatible FFS. The space allocated to TFS can be ajusted at runtime using the "tfs cfg" command in uMon[87]. With this configuration, the kernel and ramdisk images are still stored in TFS and an ASCII uMon boot script is still used to start up the application. The only addition is that now, once Linux starts up, it has access to some non-volatile area of flash for permanent storage of files. This is better than the previous example because the linux application can store non-volatile data; however, it doesn't provide any facility to allow the Linux-based application to be upgraded while Linux is running. Once again, this may or may not be ok.

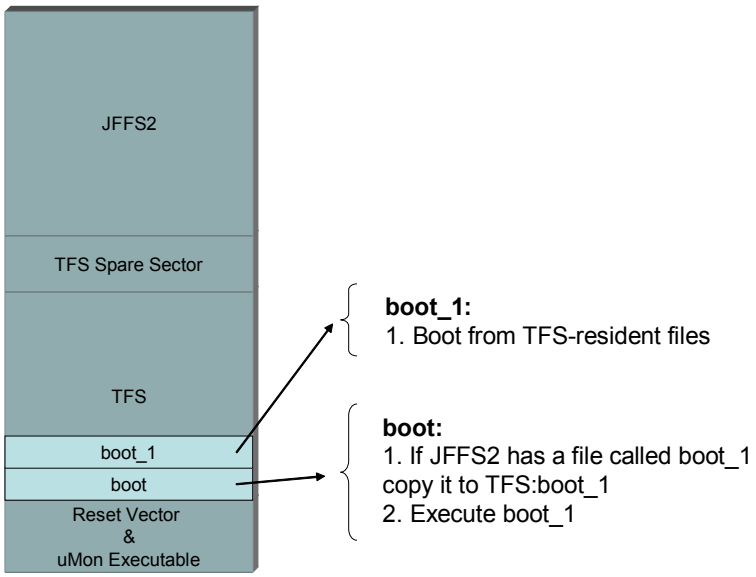


• **Figure 11: TFS & Linux FS Occupying Same Flash Device**

Best: TFS with Access to JFFS2 and/or FAT

The flash footprint may grow in complexity, but it also grows in versatility. As of release 1.9 of MicroMonitor, JFFS2 and FAT file formats are supported and can become part of the scriptable boot strategy for your system. The philosophy is the same reqardless of whether JFFS2 or FAT (or both) is used, so for the sake of this discussion we'll refer to JFFS2 and/or FAT as the "non-TFS" file system. The basic idea is quite simple: uMon has the ability to query the non-TFS FS for a specified file, plus it can copy that file from the non-TFS FS into TFS or memory. With this capability, the running application can now create a file in its native FS and at boot time, uMon can use that file to affect the restart.

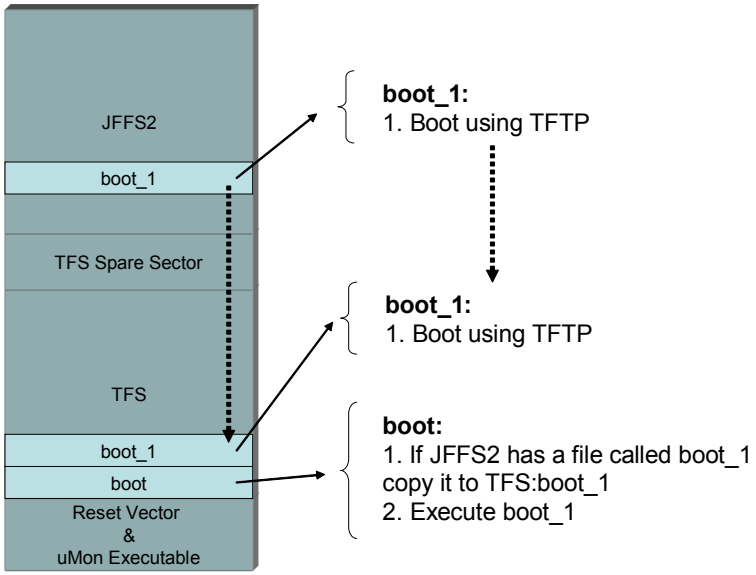
[86] It's beyond the scope of this text to discuss the actual scripts that would be used to start up a system with a Linux kernel and FS image.
[87] The "tfs cfg" capability is new as of uMon 1.8, and is port-specific, so it must be enabled at build time.



• **Figure 12: Boot With No JFFS2 Override**

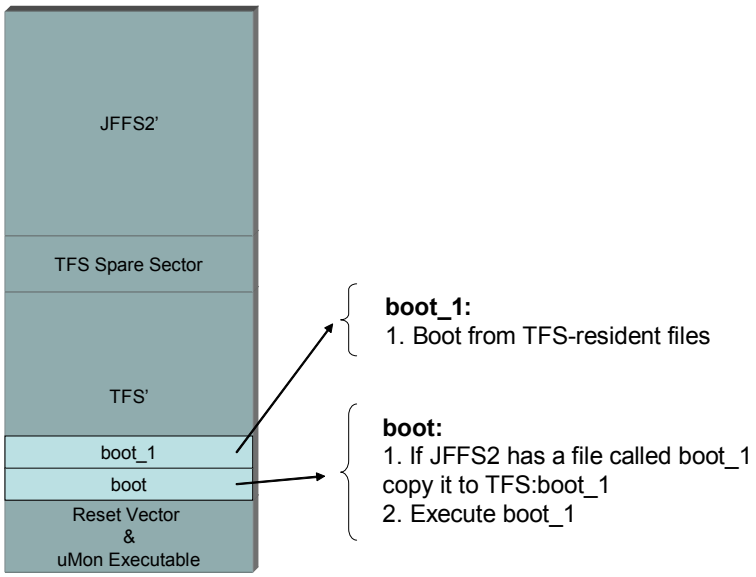
So, referring to Figure 12 as the first-time boot scenerio, notice that uMon has two scripts: <boot> and <boot_1>. The pseudo-code of <boot> shows that it simply looks to JFFS2 for a file called boot_1, and if found, it copies that file to TFS. The next step is to simply execute the boot_1 script (whether it was copied or not). In the case of Figure 12, there is no JFFS2 based override, so the <boot_1> script already installed in TFS just runs.

Now, lets assume that the application is up and running and one way or another the file <boot_1> is stored into JFFS2 at some well-known directory location. Then the board is reset.



• **Figure 13: Boot With Override**

Now, referring to Figure 13, the <boot> script runs. This time it detects the <boot_1> file in JFFS2 space, so it copies that file to TFS, then executes it. The new boot strategy calls for a TFTP transaction that may update what is in TFS or may update something in the non-TFS space (totally up to the application).



• **Figure 14: Updated Flash after Override boot_1 Script**

This update can include a new TFS-resident boot_1 script, and upon restart of the application, the boot_1 script in JFFS2 can be removed. In summary, the creation of the JFFS2-resident boot_1 script was used to affect the boot process dramatically, but quite simply. The end result (referring to Figure 14) is potentially new TFS data (TFS') and new JFFS2 data (JFFS2'), totally dependent on the application needs. Note that this works the same way for JFFS2 (board-resident NOR flash) as FAT (removable compact flash).

13.1.3 Conclusion

It's important to note that each of the above scenerios has its own set of pros and cons, so the "good/better/best" prefix isn't really appropriate simply because the added versatility can sometimes be seen as added vulnerability. Nevertheless, the point is that there are several different ways the system can be configured, maintained and updated. It is also important to note that this versatility is in no way limited to embedded Linux. The Linux application discussed above, can easily be replaced with some other RTOS or no RTOS at all, and the same flexibility provided by uMon is still available. Actually, as soon as the MMU portion of the equation is removed, things get a lot simpler. So, in the end, it all depends on what you need for your system; and regardless of the OS required, uMon supports configurability that is likely to fit.

13.2 Linux Startup Using MicroMonitor

For almost all embedded Linux systems there is the following sequence of events that must take place prior to the kernel actually running…

- System reset and basic CPU initialization (initialize cache & interrupts, etc…)
- If not on-board, then the kernel and/or rootfs images must be transferred to the target.
- If on-board, the kernel and/or rootfs may need to be copied/decompressed to RAM.
- Initialization of site dependent data (start & size of memory, console baud rate, etc…) passed from bootloader to kernel.
- Transfer of control from bootloader to kernel through the kernel's entrypoint.

Each of these is site-dependent, in other words, from one target to the next, the actual details are slightly different. Some of this just can't be avoided because different system architectures have different requirements. This section talks about some of the facilities provided by uMon to manage these differences (in some cases without having to write any specific boot code at all). The next two sections discuss port-specific commands (ldatags & lboot) in uMon that can be adapted to your hardware for booting kernels. The third section discusses a command (struct) that may be able to eliminate the majority of the port-specific code (includeing use of ldatags and/or lboot)[88] used to hook the monitor's environment up to the kernel.

13.2.1 Using 'lboot' for PPC-based Linux Startup

This command is port-specific because it has to initialize a structure, usually referred to as the "board information structure" (or bdinfo), that is very specific to the particular kernel. A typical board-info structure contains entries that tell the kernel things like memory start, memory size, PCI bus frequency, CPU internal bus frequency, ethernet address, console baudrate, and more. The lboot command simply knows about this structure, initializes it with some default values, but also supports a user interface that allows the user to modify any of the entries as needed. In some cases, lboot looks to shell variables for overriding the default values. In other cases, it uses command line arguments. This is 100% up to the writer of the port, which unfortunately implies that "no two lboot commands are exactly the same". Nevertheless, it works just fine, and realistically, there are just two or three common "themes" used when porting a new lboot command to a new board.

Usually, this command also jumps into the kernel. The calling parameters for the PPC-based kernels that I've used is:

```
kernel(bdinfo_ptr, initrd_start, initrd_size, cmdline, cmdline_len)
```

[88] The intent is that the 'struct' command will eliminate the need for 'lboot' and/or 'ldatags'; however, there is no near-term plan to remove 'lboot' or 'ldatags' from the uMon source tree. They are briefly discussed here for the sake of completeness; emphasis is clearly pointed to the 'struct' command for future ports.

The lboot simply builds the parameter list from its own internal knowledge (either defaults or user-defined on the command line) of the system and makes the jump into the kernel. That's all there is to it.

13.2.2 Using 'ldatags' for ARM-based Linux Startup

Similar to the 'lboot' command, ldatags is usually also port specific; however the nature of the "tags" structure makes it slightly more likely that the same command can be used on several ports. The idea of a tags structure is to provide some means for the receiving code to parse the incoming structure with some intelligence. Part of the structure is used to identify itself and what comes next. This allows the receiving code (in this case the kernel startup code) to intelligently look through the structure for data that may or may not be needed from one port to the next. It also allows the set of tagged entries to vary without the need for a code change as long as the tag is supported in the code. It's a good improvement over the hard-coded structure technique discussed above.

The bottom line is that ldatags also populates a structure (in this case a 'tagged' structure, nevertheless, still a structure) that is eventually read by the kernel because one of the parameters passed to the kernel's entrypoint is a pointer to this structure. As a result, the ldatags command has some hardcoded tag structure that can be modified by the command line; still if a new member of the tags structure is needed, then the ldatags command code must be modified and the monitor must be udpated to support this. Like the 'lboot' command, this works just fine and if your port already uses this command you can just stick with it if you prefer.

13.2.3 Using 'struct' for ANY CPU

If you've read the two previous sections, then hopefully you've picked up on my point. While both the 'lboot' and 'ldatags' commands work just fine. They're a coded solution for something that is much better off as a script. The advantage being that if something changes, you simply modify the script, no need to modify the monitor binary. This is where the "struct" command comes in. This command doesn't know anything about a kernel entrypoint or a board-info structure or a tags structure, it simply provides the generic ability to build ANY (not quite, but close enough for this conext) kind of structure in memory.

A picture paints a 1000 words, so let's get right to an example. If you haven't already done so, then now is the time to read through manpage for the struct command. This is an example that demonstrates how to use 'struct' as a replacement for the lboot command for configuring embedded Linux on a PowerPC. Note that there is nothing specific to Linux here, just the structure definitions. They're what would be used to establish a board information structure in memory prior to turning over control to the linux kernel. The requirement on the system is that the file vmlinux.bin.gz (compressed kernel) and initrd.img.gz (compressed ramdisk) be previously built and put into TFS on the target. The script shown in Listing 55 can be used to decompress the kernel image to its entrypoint (0x00000000), establish the board info structure and jump into the kernel passing it a pointer to the board info structure, the location and size of the ramdisk, and the begin and end of the kernel command line.

```
01: # Build the board info structure (bd_info)...
02: #
03: ###>>struct binfo {
04: ###>>    long   memstart;
05: ###>>    long   memsize;
06: ###>>    long   flashstart;
07: ###>>    long   flashsize;
08: ###>>    long   flashoffset;
09: ###>>    long   sramstart;
10: ###>>    long   sramsize;
11: ###>>    long   bootflags;
12: ###>>    long   ip_addr;
13: ###>>    char   enetaddr[6];
14: ###>>    short  ethspeed;
15: ###>>    long   intfreq;
16: ###>>    long   busfreq;
17: ###>>    long   baudrate;
18: ###>>    char   s_version[4];
19: ###>>    char   r_version[32];
20: ###>>    long   procfreq;
```

```
21: ###>>    long   plb_busfreq;
22: ###>>    long   pci_busfreq;
23: ###>>    char   pci_enetaddr[6];
24: ###>>    char   cmdline[256];
25: ###>>}
26:
27: tfs size initrd.img.gz RAMDISK_SIZE
28: tfs base initrd.img.gz RAMDISK_BASE
29: unzip vmlinux.bin.gz 0
30:
31: set STRUCTBASE 0x03000000
32: set STRUCTFILE $ARG0
33: struct binfo=0
34: struct binfo.memsize=0x08000000
35: struct binfo.intfreq=300000000
36: struct binfo.busfreq=100000000
37: struct binfo.enetaddr[6]=e2b(${ETHERADD})
38: struct binfo.baudrate=${CONSOLEBAUD}
39: struct binfo.cmdline[256]=strcpy("console=tty1")
40: struct binfo.cmdline[256]=strcat(" console=ttyS0,${CONSOLEBAUD}")
41: struct binfo.cmdline[256]=strcat(" ip=on")
42: struct binfo.cmdline[256]=strcat(" root=/dev/ram rw")
43: struct binfo.cmdline[256]=strcat(" ramdisk_size=${RAMDISK_SIZE}")
44:
45: struct binfo.cmdline[256]
46: set CMDLINE=hex(${STRUCTBASE}+${STRUCTOFFSET})
47: pm -S $CMDLINE ""
48: call 0  $STRUCTBASE $RAMDISK_START $RAMDISK_END $CMDLINE $STRLEN
49: exit
```

• Listing 55: Complete Linux Boot Script

Let's walk through the script (it is assumed that you've looked at the 'struct' manpage)… A high level view of the script above shows four main sections: the structure declaration, the structure initialization, file preparation and the transfer of control from uMon to kernel. Lines 3-25 define the board-information structure used by the kernel at startup. Notice that each line has the ###>> prefix so that the structure definition can be pulled out of the same file as the script. Lines 27-28 are used to retrieve the base address and length of the initrd.img.gz file. Line 29 decompresses the kernel image to address zero Lines 31-43 establish the structure in memory at address 0x03000000. Setting the STRUCTFILE variable (line 32) to $ARG0 (the name of the running script) is what tells the struct command to expect the leading ###>> in each line. Line 33 clears the entire structure. Lines 34-36 initialize a few structure members with fixed values. Line 37 demonstrates the use of the "e2b" (ethernet-to-binary) function on the right side of the equal sign. Line 38 is a basic initialization but uses a shell variable. Lines 39-43 demonstrate the use of strcpy/strcat functions to establish the content of an array in the structure. Lines 45-47 are used to load CMDLINE and STRLEN with the base address and length of the command line array within the structure. Last, but certainly not least, line 48 jumps into the entrypoint (0x00000000) passing the previously mentioned 5 parameters.

That's it! Aside from the basic initialization of the CPU (that would have been done by the bootloader anyway), this covers the detail of a complete kernel startup. Everything is out in the open (i.e. visable and easily changeable) and can be changed if the underlying kernel changes.

13.3 Using JFFS2 and/or FATFS as Part of Your Startup Strategy

If JFFS2 or FATFS is used by your Linux port, then it's quite possible that your port will have the jffs2 and/or fatfs command available to uMon as well. With this capability, the linux environment can even have its own kernel and initrd files within its own file system. They are accessible by the boot monitor hence, the boot monitor can extract them from the FS and boot from them similar to the way it would boot if the images were stored in TFS. The advantage here is that this makes it very easy to update a kernel or initrd image when running linux. The disadvantage here is that this makes it very easy to delete a kernel or initrd image when running linux. That's a design decision, and ultimately it just depends on what is in your system. Perhaps, the boot logic is set up to use those JFFS2/FATFS files only if they are installed. If not, then boot from TFS or TFTP or some other means.

```
jffs2 qry vmlinux.bin
if $JFFS2TOT eq 1 goto KERNEL_IN_JFFS2
if $JFFS2TOT eq 0 goto KERNEL_IN_TFS
echo Should not get here!
exit

# KERNEL_IN_JFFS2:
jffs2 get  vmlinux.bin 0x0
goto KERNEL_AT_ZERO

# KERNEL_IN_TFS:
unzip vmlinux.bin.gz 0x0
goto KERNEL_AT_ZERO
```

• Listing 56: Query JFFS2 for New Kerenl

The script of Listing 56, is just an example. The FATFS command would use the same type of logic. The point is that this is all quite scriptable to a logical flow that meets the needs of your system. Refer to the man pages for each of the commands for more details.

13.4 Using the 'tfs' Command at the Linux Prompt

We've mentioned the 'jffs2' and 'fatfs' commands for use in uMon to allow it to query the file system used by linux. This provides some flexibility at boot time. As an alternative, the 'tfs' command built to run in linux provides some ability to read/write/query the content of TFS when Linux is running. This command assumes that some MTD partition covers the area that is used by TFS, so this command does require some kernel modification to work properly. Listing 57 shows an example output of "cat /proc/mtd"…

```
linux# cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00100000 00020000 "Boot Image"
mtd1: 00800000 00020000 "Raw Block"
mtd2: 03000000 00020000 "JFFS2 FS1"
mtd3: 03000000 00020000 "JFFS2 FS2"
mtd4: 00400000 00020000 "TFS"
linux#
```

• Listing 57 : Output of "cat /proc/mtd"

The nice thing about this command is that once the TFS space is mapped into MTD within the kernel (simple addition to the map already established in your system), any file within any FS accessible by Linux can be transferred into TFS (assuming the space is available of course). The command provides the ability to list the files in TFS, retrieve a file from TFS (to Linux) or put a file into TFS (from Linux).

```
linux# tfs
Usage: tfs [options] {infile} {command} [command args]
Options:
   -h              generate this help message
   -M {path}       specify MTD path (default="/dev/mtd")
   -m {fname}      specify MTD proc file name (default="/proc/mtd")
   -t {pname}      specify TFS partition name (default="TFS")
   -v              additive verbosity

Commands:
   ls                    :  list all files in TFS partition
   ls {filename}         :  list specified file and load exit status
                            (1 = file not found, 0 = file found)
   dump                  :  dump headers of files in TFS partition
   get {tfile} [lfile]   :  copy TFS file to Linux file
   put {lfile} [tfile]   :  copy Linux file to TFS file
   rm {tfile}            :  remove file from TFS
linux# tfs ls
 boot_script
```

```
 monrc
 vmlinux.bin.gz
 initrd.img.gz
linux#
```

• Listing 58: Linux 'tfs' Command Usage

The one limitation here is that this does not support any TFS defragmentation, so if there is no space at the end of TFS's flash area, a file can't be copied from Linux. This turns out not to really be a problem because typically this command would only be used to modify some configuration file or boot file in TFS anyway (i.e. a small file), so the target's startup script can contain logic that will automatically invoke a defragmentation at startup if there's not some minimum amount of free TFS space available. For example, assuming we want to defrag TFS if there isn't at least 64K of free space. We would do this within some startup script (not monrc), using the following lines…

```
tfs –d //FLASH/ freemem TFSFREESPACE
if $TFSFREESPACE ge 0x10000 goto SKIP_CLEANUP
tfs –d //FLASH/ clean
# SKIP_CLEANUP:
```

• Listing 59: Auto TFS Defragmentation if Free Space is Below 64K

The "-d //FLASH/" option in Listing 59 is only necessary if there is more than one TFS partition. In this case the partition we are dealing with is //FLASH/. Even if there's only one TFS partition, this is still legal. Refer to the output of "tfs stat" to determine the partition names of your port.

Chapter 14 Shell Variables Created and/or Used by MicroMonitor

When the monitor starts up, it looks for the presence of certain shell variables to configure itself (see sections 3.2.2 & 3.2.3 as examples); also, it sets up some shell variables to be used by scripts or application code for various purposes. The *set* command supports the ability to assign values to variable names and perform some basic operation on them. Many other commands within the monitor use and/or create shell variables. Within MicroMonitor, shell variables are global. If a script uses the *set* command or an application uses mon_setenv() to create a shell variable then exits, that shell variable will still be available to the system. Refer to section 3.4 for details on command line syntax. Following is a list of the shell variables intrinsic to the monitor:

14.1 APPRAMBASE

This shell variable is loaded with the starting point of the RAM space that is made available to the application. This typically starts on some modulo 0x1000 boundary just above the end of RAM space that is used by the monitor itself. Note that this variable is automatically loaded by the monitor at startup. Certain facilities within the monitor use this value as a pointer to memory that is assumed to be accessible. This is important to be aware of, so here are the facilities that make this assumption: edit, tftp server, tfs cp, xmodem receive, and the non-power safe version of tfsclean. If your application will use these facilities at runtime, then the application must be mapped somewhere above the APPRAMBASE address so that these other facilities will not overwrite the application space. The value of the shell variable can be modified, and the modified value will then be used by these facilities, but make sure you know what you're doing! In general, its intent is to provide a common means of accessing the address that has been loaded by the monitor.

14.2 ARGC

Argument count. This variable is automatically loaded with the current argument count when a script is run. The count includes argv[0] as an argument, so in the following script (named argc_test)

```
echo ArgCount = $ARGC
echo Arg0 = $ARG0
```

invocation of “argc_test” at the command line will result in a $ARGC value of 1. If the command line was “argc_test abc def ghi”, then the $ARGC value would be 4.

14.3 ARG'N'

Argument content. The shell variables ARG0 thru 'N' are automatically loaded with the argument list when a script is run. Following the above example, each argument on the command line is loaded into the ARGN shell variable, so $ARG0 would contain “argc_test”, $ARG1 would contain “abc”, etc. Note that since the shell variables within MicroMonitor are global, these ARGN variables will still exist after the script terminates.

14.4 APP_EXITONCLEANERROR

This shell variable is used for testing TFS. It can be set to some value (anything, as long as it is set), and this will cause any error in defragmentation to result in a call to mon_appexit(0). Useful when testing for defragmentation faults. See also: SCR_EXITONCLEANERROR (section 14.60 below).

14.5 ARPRETRYTUNE

If this shell variable is set, then the ARP retry mechanism is reconfigured to the values specified. The format of the content of this variable is XX:YY:ZZ; where XX is the retransmit_delay value, YY is the giveup_count and ZZ is the retransmit_delay_max value. If the shell variable is not set, the defaults are 1:0:4. Refer to discussion of the tftp command (section 15.40) for more details.

14.6 BOOTFILE

This variable is loaded by the monitor's BOOTP/DHCP clients based on the content of the "file" member of the BOOTP/DHCP response. Refer to RFC 951 (bootp) or RFC 2131 (dhcp).

14.7 BOOTROMBASE

This variable contains the address that the monitor sees as the starting point of the base flash device. This is typically used to allow a host-based script to be unaware of where the actual starting point of the flash is,

but still transfer data to it. Note that the monitor at startup automatically loads this variable. The user can modify it, but its intent is to provide a common means of accessing the address that has been loaded into it by the monitor; so, assume it is read-only.

14.8 BOOTSRVR

This variable is loaded by the monitor's BOOTP/DHCP client based on the content of the "siaddr" member of the BOOTP/DHCP response. Refer to section 6.1.

14.9 CF_BLKSIZE

This variable is automatically loaded with the blocksize used by the 'cf' (compact flash) command when 'cf init' is called.

14.10 CMDSTAT

This variable is loaded with the status of the previous command within a script. Note that this is only populated by the command if it is within a script. The value will be either "PASS" or "FAIL".

14.11 CONSOLEBAUD

This variable is used to allow an application to run with the same baud rate that the monitor is currently running at, plus it can be set in the monrc to override the default console baud rate used by the monitor. At initial startup, the monitor configures its COM port to some pre-defined baud rate (hard-coded when the monitor was built for the target), then after the monrc file is run, the monitor looks for the presence of the CONSOLEBAUD shell variable. If set, the console baud rate is automatically set to the value stored in CONSOLEBAUD. If not set, then the monitor sets this variable to the value configured when the monitor was built. In either case, this variable is accessible by the application so that the application can use it if it re-configures the serial port so that the baud rate used by the application will match the baud rate used by the monitor.

14.12 DCLIPORT

If this shell variable is present at startup, then MicroMonitor's DHCP client will use this as the client port number instead of the default of 68.

14.13 DHCPDONTBOOT

If this shell variable is present, then when uMon runs DHCP (or BOOTP), if a bootfile is present in the transaction, the file will be downloaded from host to target ram (starting at $APPRAMBASE); however, it will not be transferred to TFS and the content of that data will not be executed. It will be up to the user (or script) to deal with the downloaded data whose size is stored in $TFTPGET.

14.14 DHCPCLASSID

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, it will include this string as the DHCP Extension "Class-identifier" (refer to RFC 2132). The format of the content of this variable is a simple string.

14.15 DHCPCLIENTID

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, it will include this string as the DHCP Extension "Client-identifier" (refer to RFC 2132). The format of the content of this variable, since it represents a type:value pair, is #:HHHHHH...; where '#' is a decimal value less than 256, ':' is a delimiter, and 'HHHH...' is a string of ASCII-coded pairs where each 'HH' is converted to one 8-bit binary value.

14.16 DHCPFLAGS

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, or the BOOTP client issues a BOOTP_REQUEST, it will use the value stored here as the flags. If not present, then the flags are zero. Note that the only valid bit at this point is 0x8000 (enable broadcast reply), and it is only applicable to DHCP.

14.17 DHCPLEASETIME

This shell variable serves two purposes for the DHCP client :

- If set in monrc, then it will be used as a minimum requirement lease time if the server sends the client a lease time (if the lease time from the server is less than the content of this shell variable, then ignore the server's offer);
- It will be loaded with the actual lease time (if any) specified by the server. The value is in hex, and will only be present if the server has sent the client some minimum lease time.

14.18 DHCPOFFRFLTR

If this shell variable is present when MicroMonitor's DHCP client receives a DHCP_OFFER, the client will look to the content of DHCPOFFRFLTR to determine if the offer should be accepted or ignored. The format of the content of this variable is

DHCP_FIELD_ID,EXPECTED_STRING

where...

DHCP_FIELD_ID can be:

BFN:	bootfile name
SHN:	server-host name
VSO###	vendor-specific option number embedded within Vendor Specific Information (opt#43) (valid range: 0 < opt < 255)
SSO###	site-specific option number (valid range: 127 < opt < 255)

and...

EXPECTED_STRING is the ASCII string that must be within the specified field. Note that "within" means that the string may be the entire string returned by the server or a sub-string within the string returned by the server. Refer to the logic in ValidDHCPOffer() (MicroMonitor source code) for complete details.

14.19 DHCPRETRYTUNE

If this shell variable is set, then the DHCP retry mechanism is reconfigured to the values specified. If the shell variable is not set, the defaults are 4:6:64. Refer to description of ARPRETRYTUNE (section 14.5) for syntax.

14.20 DHCPRQSTLIST

If this shell variable is present, then if the MicroMonitor's DHCP client issues a DHCP_DISCOVER, it will include the digits extracted from this string as the DHCP Parameter Request List (refer to RFC 2132 section 9). The format of the content of this variable is #:#:#:#...; where each '#' is a decimal value representing one of possibly several option requests to be made.

14.21 DHCPSTARTUPDELAY

If present, this value will override the default random startup delay used by the DHCP client. By default, the DHCP transaction is delayed (out of reset) by a value (in seconds) that is relative to the least-significant byte of the board's MAC address. This is done to support some random time (between 1 & 10 seconds) for DHCP startup in cases where several systems are reset simultaneously. To override this random delay, set this shell variable to the number of desired seconds of delay (including zero).

14.22 DHCPVSA

This shell variable will be loaded with an ASCII-coded hex string that represents the vendor-specific area returned by the BOOTP or DHCP server. For BOOTP this array is fixed at 64 bytes of binary data, so DHCPVSA will contain a 128-byte ASCII-coded-hex copy of that array. For DHCP the size of the vendor-specific area can vary. Since the monitor scans for options itself, the function DHCPGetOption() can be re-used in application space after converting the content of DHCPVSA to binary. Note that this variable will only be populated if it exists prior to the DHCP/BOOTP transaction; so, if you want the transaction to save the VSA data, set DHCPVSA to TRUE prior to starting the transaction.

14.23 DSRVPORT

If this shell variable is present at startup, then MicroMonitor's DHCP client will use this as the server port number instead of the default of 67.

14.24 ENTRYPOINT

This variable is set by the tfs ld command. The value corresponds to the entrypoint address of the application just loaded. It a script to call "tfs ld", then "call $ENTRYPOINT" as a verbose alternative to just running the executable.

14.25 ETHERADD

This variable is expected to contain the MAC address that is to be assigned to the target. The MAC address can be retrieved in one of several different ways. Refer to Listing 42 for details.

14.26 EXCEPTION_SCRIPT

If set, then the content of this shell variable is assumed to contain a script name that is to be executed when an exception occurs. Refer to Listing 30 for details.

14.27 EXCEPTION_TYPE

If an exception occurs, this shell variable will contain the type of exception that occurred. Useful for logging to a file at the time of an exception. The actual string populated into this variable is port and CPU specific.

14.28 FATFS_RD

Used by the fatfs command to contain a hex address that represents the address of the blockread() function that will be used by fatfs to retrieve data from some block-storage interface. Refer to fatfs command for more details.

14.29 FATFS_WR

Used by the fatfs command to contain a hex address that represents the address of the blockwrite() function that will be used by fatfs to write data to some block-storage interface. Refer to fatfs command for more details.

14.30 FATFSNAME

Loaded with the name of the file listed by most recent "fatfs qry {filename}" command (note that the 'filename' argument may contain a wildcard. Refer to fatfs command manpage for more details.

14.31 FATFSSIZE

Loaded with the size of the file listed by most recent "fatfs qry {filename}" command. Refer to fatfs command manpage for more details.

14.32 FATFSTOT

Loaded with the total number of files listed by the most recent "fatfs ls {filename}" command. Refer to fatfs command manpage for more details.

14.33 FLASH_BASE_N

Loaded after the "flash info" command is run. This variable contains the base address of flash device 'N' (where 'N' starts at zero). Refer to the flash command manpage for more information.

14.34 FLASH_SCNT_N

Loaded after the "flash info" command is run. This variable contains the total number of sectors of flash device 'N' (where 'N' starts at zero). Refer to the flash command manpage for more information.

14.35 FLASH_END_N

Loaded after the "flash info" command is run. This variable contains the end address of flash device 'N' (where 'N' starts at zero). Refer to the flash command manpage for more information.

14.36 FLASH_DEVTOT

Loaded after the "flash info" command is run. This variable contains the total number of flash devices in the system. Refer to the flash command manpage for more information.

14.37 GDBPORT

If this shell variable is present at startup, then MicroMonitor's GDB UDP port is set to this value instead of the default value of 1234.

14.38 GIPADD

The Gateway IP address. This variable must be established for systems on a network. It is automatically loaded by BOOTP/DHCP from the vendor-specific options if applicable. If BOOTP/DHCP is not used, then it should be established by the execution of the monrc file.

14.39 IPADD

This variable is expected to contain the IP address that is to be assigned to the target. The IP address can be locally configured in one of several different ways (refer to Listing 42 for details). It can also be established through DHCP/BOOTP.

14.40 JFFS2NAME

Loaded with the name of the file listed by most recent “jffs2 qry {filename}” command (note that the ‘filename’ argument may contain a wildcard. Refer to jffs2 command manpage for more details.

14.41 JFFS2SIZE

Loaded with the size of the file listed by most recent “jffs2 qry {filename}” command. Refer to jffs2 command manpage for more details.

14.42 JFFS2TOT

Loaded with the total number of files listed by the most recent “jffs2 ls {filename}” command. Refer to jffs2 command manpage for more details.

14.43 MALLOC

The heap -m command returns a block of memory from the monitor's heap. It also populates this shell variable with the address of the block.

14.44 MCMDPORT

If this shell variable is present at startup, then MicroMonitor's MONCMD server will use this as its port number instead of the default of 777.

14.45 MEMSIZE

This variable is loaded by the “mt” command when the –S option is used to determine the size of memory.

14.46 MONCMD_SRCIP

This shell variable is populated by MicroMonitor whenever an incoming ‘moncmd’ message is received. It contains the IP address of the device that issued the remote command.

14.47 MONCMD_SRCPORT

This shell variable is populated by MicroMonitor whenever an incoming ‘moncmd’ message is received. It contains the port number of the device that issued the remote command.

14.48 MONCOMPTR

This shell variable is populated by MicroMonitor at bootup with the location of the well-known address used to hook the application code to the monitor’s API.

14.49 MONFLAGS

If this shell variable is set at startup, the monitor will use it to setup some internal flags. The format of the content of this variable is xx:yy:zz...; where "xx", "yy" and "zz" are ASCII strings (flag names) that represent the setting of certain binary flags in the monitor...

FLAG NAME	FLAG MEANING

nophdr	do not print initial header when monitor is reset
nopdf	quiet defragmentation when verbosity is off
noptftp	quiet TFTP when verbosity is off
nopmcmd	quiet MONCMD when verbosity is off
notftpovw	do not allow incoming TFTP write requests to overwrite an existing file
noexitstat	supress the “Application Exit Status (X)” message generated by mon_appexit

Note that this shell variable is only read at system startup. Changing it in runtime has no affect on the internal flag settings.

14.50 MTCRC

This variable is loaded by the –C option of the ‘mt’ command, which does a CRC32 calculation over the specified range of memory.

14.51 MONITORBUILT

This variable contains the string created by the concatenation of the following three strings at monitor build time: __DATE__ “ @ “ __TIME__

14.52 NETMASK

This variable should contain the mask of the subnet on which the target resides. This variable must be established for systems on a network. It is automatically loaded by BOOTP/DHCP from the vendor-specific options if applicable, or it should be configured by the execution of the monrc file.

14.53 NO_EXCEPTION_RESTART

If this shell variable is present, then at the time of an exception, the monitor will not reset. The system will remain at the monitor level. Refer to the discussion of section 10.5.

14.54 NO_UMONBSS_WARNING

By default, the TFS loader will automatically check to see if the load destination is in BSS space that is owned by the monitor. If this is undesirable, then set this shell variable to TRUE.

14.55 PCISIZE

Loaded by the “pci size” command with the size of the area associated with the specified BAR.

14.56 PLATFORM

Every monitor is built with a platform name. This is just a verbose description of the target for general use by the monitor and/or application. It is set at monitor build time from the content of the PLATFORM_NAME definition in the config.h file.

14.57 POLLTIMEOUT

This variable can be set to override the default of ~2 seconds of polling waiting for a response when a file is autobooting with query enabled.

14.58 PROMPT

This variable contains the string that the monitor uses as the user prompt. If not set, the default prompt is uMON> and this is loaded into the PROMPT shell variable. At any time this shell variable can be changed and the prompt used by the monitor will change to the content of PROMPT.

14.59 RLYAGNT

The Relay Agent IP address. It is automatically loaded by BOOTP/DHCP from the giaddr field of the BOOTP/DHCP response if applicable.

14.60 ROOTPATH

This variable is populated by the content of DHCP option #17 if the incoming DHCP message contains option 17 (root path).

14.61 SCR_EXITONCLEANERROR

This shell variable is used for testing TFS. It can be set to some value (anything, as long as it is set), and this will cause any error in defragmentation to result in setting the internal flag that causes a script to terminate. Useful when testing for defragmentation faults. See also: APP_EXITONCLEANERROR

14.62 SCRIPT_IGNORE_ERROR

If this variable is present, then a running script will not stop when a command line error is detected.

14.63 SCRIPTVERBOSE

This variable, if present, the level of verbosity to be used during script execution. Valid values are 0, 1 & 2. 0 is no verbosity, 1 means the command line is echoed, 2 means the command line is echoed before and after CLI processing. This shell variable is tested prior to each line of the script execution. If set, then it is used as the verbosity level; if not set, then the default level is used. Note that this variable can be changed within a script to provide different levels of verbosity at different points in the script.

14.64 STRLEN

This variable is created by the –s and –S options of the pm command. It is loaded with the length of the string created by either “pm –s” or “pm –S”. See the pm command for more details.

14.65 STRUCTBASE

Used by the ‘struct’ command as the base address of the referenced structure. Refer to the struct command manpage for more details.

14.66 STRUCTFILE

If this shell variable is set, then the default file name of "structfile" is overridden with the content of this variable. See discussion on the use of the file named “structfile” in section 10.2.2 above. As of uMon1.9, this variable is used by both the ‘cast’ and ‘struct’ command.

14.67 STRUCTOFFSET

Set by the ‘struct’ command to the distance between the specified member and the base of the specified structure (offset).

14.68 STRUCTSIZE

Set by the ‘struct’ command to the size of the specified structure or structure member.

14.69 SYMFILE

If this shell variable is set, then the default file name of "symtbl" is overridden with the content of this variable. See discussion of symbols (section 3.4).

14.70 TFTPGET

If this shell variable is present after a *tftp get* command, it is indication that the file transfer succeeded; it will then contain the number of bytes of data that have been transferred through tftp.

14.71 TFS_PREFIX_N

Populated by the “tfs stat” command. This variable contains the prefix used by TFS partition ‘N’, where ‘N’ starts from zero.

14.72 TFS_START_N

Populated by the “tfs stat” command. This variable contains the start address of TFS partition ‘N’, where ‘N’ starts from zero.

14.73 TFS_END_N

Populated by the “tfs stat” command. This variable contains the end address of TFS partition ‘N’, where ‘N’ starts from zero.

14.74 TFS_SPARE_N

Populated by the “tfs stat” command. This variable contains the start address of the spare sector used in TFS partition ‘N’, where ‘N’ starts from zero.

14.75 TFS_SPARESZ_N

Populated by the “tfs stat” command. This variable contains the size of the spare sector used in TFS partition ‘N’, where ‘N’ starts from zero.

14.76 TFS_SCNT_N

Populated by the “tfs stat” command. This variable contains the number of sectors used by TFS partition ‘N’, where ‘N’ starts from zero.

14.77 TFS_DEVINFO_N

Populated by the “tfs stat” command. This variable contains the content of the device information field used by TFS partition ‘N’, where ‘N’ starts from zero.

14.78 TFS_DEVTOT

Populated by the “tfs stat” command. This variable contains the total number of TFS partitions currently established.

14.79 TFTPPORT

If this shell variable is present at startup, then MicroMonitor's TFTP client/server will use this as its port number instead of the default of 69.

14.80 TFTPRCV

After a file is received via uMon’s TFTP server, the size of that file transfer is logged to this shell variable.

14.81 TFTPRETRYTUNE

If this shell variable is set, then the TFTP retry mechanism is reconfigured to the values specified. If the shell variable is not set, the defaults are 4:3:32. Refer to description of ARPRETRYTUNE for syntax.

14.82 VERSION_MAJ, VERSION_MIN, VERSION_TGT

These three variables are automatically initialized with the version number of the monitor. The version number is a 3-field, value (X.Y.Z) that contains the major (X) and minor (Y) release number and a port-specific release number (Z). For example, assume we are referring to version X.Y.Z of the Cogent CSB360 port, this would be stated as “CSB360 port release Z of uMon version X dot Y”

14.83 XMODEMGET

Populated with the size of the most recent Xmodem command transfer.

Chapter 15 MicroMonitor Command Set

MicroMonitor has a large command set. The majority of commands are universal to all target systems; however, being an embedded system platform, there are obviously some unique features of various CPUs and target hardware that call for unique commands. This section describes each of the target-independent commands. There are a few main categories of commands…

Memory display, modification & test:
CAST, CM, DM, EDIT, FLASH, FM, MT, PM, SM, STRUCT

Network interface:
ARP, DHCP, ETHER, ICMP, TFTP, SYSLOG

File/data transfer:
TFTP, XMODEM

Script specific:
ECHO, EXIT, GOSUB, GOTO, IF, ITEM, READ, RETURN, SET, SLEEP

Application debug:
CALL, DIS, GDB, MTRACE, PROF, STRACE

Miscellaneous:
HELP, HEAP, HISTORY, RESET, VERSION, ULVL, CF, SD

File specific:
TFS, JFFS2, FATFS, UNZIP

Many of the command usage lines are displayed using a combination of brackets [] and braces {}. In general, text within brackets (e.g. [arg0]) means that the text is optional and text within braces (e.g. {cmd}) means that the text is required. Also, for specifying options, the "getopt" dash syntax is used (as is common in Unix commands), and the usage summary for options will typically read "-[abcd:e:f]" meaning that options -a, -b, -c and –f are standalone and options –d and –e require arguments. If an option requires an argument that argument will be discussed within the command help text.

All command arguments in the monitor are assumed to be of the syntax requiring that hex data be entered with a leading 0x, octal with a leading 0 else assume decimal.

15.1 ARP

Arp cache maintenance..

USAGE:

arp -[fps:v] [IP]

DESCRIPTION:

ARP (address resolution protocol) is used to allow a network element to query the network for the Ethernet address assigned to a known IP address. This command is a user interface hooked into the monitor's basic ability to query the network for Ethernet addresses and maintain a cache of the most recently used addresses.

OPTIONS:

- -f
  Flush the current arp cache;
- -p
  Do not use the gateway if the IP is on a different subnet;
- -s {etheradd}
  Store the Ethernet and IP combination into the cache. For this option the IP argument is required.
- -v
  Verbose mode

EXAMPLES:

- arp 135.3.94.136
  Returns the Ethernet address for the network device that is assigned IP address 135.3.94.136.
- arp -f
  Flushes all entries in the arp cache.
- arp –s 11:33:55:88:99:AA 192.168.1.99
  Stores the arp entry of 11:33:55:88:99:AA with 192.168.1.99 into the cache.
- arp
  Returns the current content of the arp cache.

NOTES:

- If the IP address specified is not on the same subnet as the target making the query, then the actual Ethernet address that is loaded into the arp cache is that of the gateway.
- This command requires some network knowledge. In particular, it assumes that the shell variable NETMASK is loaded with the correct sub-net mask setting for the network the target is residing on. If it is determined that the IP request is for a device not on the local sub-net, then the shell variable GIPADD must be loaded with the IP address of the gateway and the arp request is destined for the gateway. If NETMASK is not set, then the IP address is assumed to not exist on the same subnet. If GIPADD is not set, then the ARP request is sent to the IP address specified (may be picked up by the gateway, if it supports proxy arp). Note that the GIPADD is only needed if '-p' is not specified.

RUNTIME EXAMPLE:

```
uMON>arp –f                          << Flush the arp cache
uMON>arp                             << After flush, 'arp' returns nothing.
uMON>arp 192.168.1.100               << Arp a local host and note the response.
00:e0:18:97:62:e4 = 192.168.1.100
uMON>arp                             << Now 'arp' dumps the content of the cache.
00:e0:18:97:62:e4 = 192.168.1.100
```

15.2 BRDINFO

Dump the board information structure (if included in the port).

USAGE:

brdinfo

DESCRIPTION:

If the monitor is built with INCLUDE_BOARDINFO set to 1, then a board information structure is established in a sector of flash that is not used by TFS. This allows the target to be initialized with some environmental (board-dependent) information that is outside of TFS (hence, not as easy to change). This command simply dumps the content of that structure.

15.3 CALL

Execute an embedded function somewhere in the memory image.

USAGE:

call -[Aaqv:] {address} [arg1] ... [argN]

DESCRIPTION:

This command allows the user to execute a function at a raw memory address. A maximum of 7 arguments can be specified. By default the function is called with the arguments converted to hex. For example call 0x12345 45 99 would be used to interface to a function located at address 0x12345 whose prototype is func(int val1, int val2). The -a option allows *call* to work with functions whose prototype is (int argc,char **argv). In this case, the –a option tells *call* to build an argument list and count that is then passed to the function.

OPTIONS:

- -A
  The parameters are built into the argument list used by the uMon API function mon_getargv().
- -a
  Pass parameters to embedded function as (int argc,char **argv)
- -q
  Quiet mode, do not print the “Returned XX” string at the end of the command
- -v {varname}
  Place the hex return value of the function call into the shell variable "varname".

EXAMPLES:

- call 0xa0041000 0x100 45 0x999
  Pass control to a function located in memory at address 0xa0041000 and it will receive (0x100,45,0x999) as its parameter list.

NOTES:

- There is no attempt to verify that the address specified has valid code. It is up to the user to make sure the address is correct and the function resides there.

RUNTIME EXAMPLE:
Assume the function…

```
int
func(int i, int j, int k)
{
       return(i + j + k);
}
```

resides in memory at 0x20054…

```
uMON>call 0x20054 1 2 3             << Function is called with three args.
Returned: 6 (0x6)                   << Returned value is the sum of the three.
uMON>
```

15.4 CAST

Overlay a structure definition onto a block of memory and display it.

USAGE:

cast -[al:n:pt:] {structure name} {address in memory}

DESCRIPTION:

This command allows the user of the monitor to display a block of memory as a structure, table of structures or linked list of structures. The definition of the structure is assumed to be in the TFS file called "structfile" (or this can be overridden by the content of the STRUCTFILE shell variable), and any number of structure definitions can be in that file. The format of the structure definition is similar in syntax to that of standard 'C' (see below).

OPTIONS:

- -a
  Display the address of each member of each structure;
- -l {linkname}
  Treat the specified address as the base of a linked list and get the next link from the member of the structure that matches the string 'linkname';
- -n {structure name}
  Specify the name of the structure to be displayed (not needed, but provides a nicer display output);
- -p
  Display any padding that is part of the structure definition;
- -t {table name}
  Treat the specified address as the base of a table and query the user for permission to display each successive entry in the table of structures.

SYNTAX OF "structfile":

To cast a structure over a block of memory, there must be some structure definition. The definition used by the cast command is assumed to be in the file "structfile". In general, the format of the entries in this file is similar to that of standard 'C' structure definitions, but with some limitations and extensions. The types "char", "short" and "long" are supported and they are displayed as a 1,2 or 4-byte decimal integer respectively. To support the ability to display in hex, the types "char.x", "short.x" and "long.x" are supported, and, if a character is to be displayed as a character (hex 0x31 printed as '1'), the "char.c" type can be applied. For example, in the following structure definition...

```
struct abc {
    long i;
    long.x j;
    char.c c;
    char.x d;
    char e;
}
```

The member 'i' would be displayed in decimal format and 'j' would be displayed in hex. The member 'c' would be displayed as a character, 'd' would be displayed in hex, and 'e' would be displayed as a 1-byte decimal integer. If a structure has an array in it, then the user must define that as an array of one of the fundamental types described above with the appropriate size. This cast command does not display arrays within structures simply because of the complexity of the output generated, so it is treated like padding with only the name and array size displayed.

Here is an example of a structure definition file that demonstrates all of the functionality of the cast command. Note that the '#' sign signifies a comment.

```
# Structures abc and def:
struct abc {
    long i;
    char.x c1;
```

```
    pad[3];            # Invisible unless -p specified.
    struct def d;
}
struct def {
    short s1;
    long ltbl[6];      # Array of 6 integers (not dumped)
    short s2;
}
```

Notice the embedded structures, use of the '.x' suffix and the pad[] format. Also, be aware that this cast command is totally unaware of compiler-specific padding and cpu-specific alignment requirements. If the structure definition puts a long on an odd boundary and the CPU does not support that, then cast will generate an exception itself. The user must add the appropriate padding to deal with this. As a result, the "pad[]" descriptor above is used for cpu/compiler-specific padding.
If the member is of type "char.c *" or "char.c []", then cast will display the ASCII string (if you don't want it to be dereferenced, use "char.x *"). Finally, do not "test" this command's parsing ability. Put one member on each line of the file and keep it syntactically simple; otherwise, you will break it!

EXAMPLES:

- cast struct_x 0x10010000
  Display memory at location 0x10010000 as if it was the structure "struct_x" as defined in the TFS file "structfile"
- cast -a struct_y 0x4000
  Display memory at location 0x4000 as if it was the structure "struct_y" as defined in the TFS file "structfile". Prefix the display of each member with the address at which that member is being retrieved from memory.
- cast -t tasktbl tstruct 0x4100100
  Starting at location 0x4100100, overlay the structure "tstruct" onto memory. Repeat the display process at what would be considered the next table entry (based on the size of the tstruct structure).
- cast -l next list 0x1000
  Assume a structure of type "list" exists at location 0x1000, then use the content of the "next" member to determine the location of the next structure to display (a linked list).

RUNTIME EXAMPLE:
Using the above structures for the content of the structfile, with the following memory at location 0x1c000…

```
0001c000: 00 00 00 2d 99 00 00 00   01 00 00 00 00 00 00 00   ...-............
0001c010: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c020: 00 00 10 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c030: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
```

the command line “cast –a abc 0x1c000” returns…

```
uMON>cast -a  abc 0x1c000
struct abc @0x1c000:
  0x0001c000: long     i: 45
  0x0001c004: char.x   c1: 0x99
  struct def d:
    0x0001c008: short    s1: 256
    0x0001c00a: long     ltbl[6]
    0x0001c022: short    s2: 4096
```

15.5 CF

Compact Flash Interface.

USAGE:

cf [options] {operation} [args]

DESCRIPTION:

This command provides some command level access to a compact flash device. This command can be used in a "standalone" mode to provide the command line interface with the ability to transfer raw blocks of data to/from the compact flash; however, as of this writing, it's real intent is to be used with FATFS (and possibly other FS commands in the future). It provides the following capabilities…

- transfer blocks of data from memory to compact flash
- transfer blocks of data from compact flash to memory
- initialize the interface and display format information
- connect its block read and block write functions to other FS facilities (i.e. as of this writing, fatfs) within uMon

As already mentioned, the primary use of this command is in coordination with the "fatfs" command; however, that isn't an absolute requirement. The raw read and write functionality may be of some use as well. The linkage to fatfs is done by the "init" operation. The shell variable prefix provided on the command line ends up being used as a prefix for a set of shell variables, XXX_RD and XXX_WR, where 'XXX' is the prefix specified. Those shell variables are loaded with the address of the compact flash's block read function (_RD) and block write function (_WR).

The fatfs command then uses the content of the FATFS_RD and FATFS_WR shell variables as pointers to blockread() and blockwrite() functions needed by the interface-independent fatfs command. It doesn't care what the underlying interface is, it just needs to know how to read a block and write a block. The "cf" command provides that hookup (as could other commands like "cf" that may also provide a block read/write interface). Refer to the fatfs command for additional information.

OPTIONS:

- -v
  additive verbosity;

OPERATIONS:

- init [shell var prefix]
  Initialize the interface and load shell variables with the read and write function pointers.
- read {destination address} {block number} {total number of blocks to be read}
  Transfer some number of blocks of data (1 block = 512 bytes) from compact flash to memory.
- write {block number} {source in memory} {total number of blocks to be written}
  Transfer some number of blocks of data (1 block = 512 bytes) from memory to compact flash.

EXAMPLE USAGE:

```
uMON>cf init FATFS   # Initialize CF interface and hook it into the fatfs command.
uMON>fatfs ls        # With FATFS_RD and FATFS_WR set, fatfs can access compact flash.
```

15.6 CM

Copy memory from one target address to another.

USAGE:

cm -[24fv] {source address} {destination address} {count}

DESCRIPTION:

Copy memory from one location in memory space to another. The size of the copy is specified by the count, which is always considered a byte-count.

OPTIONS:

- -2
  assume the width of the memory accesses is x2;
- -4
  assume the width of the memory accesses is x4;
- -f
  assume the destination address is a FIFO;
- -v
  verify that 'count' bytes of data starting at the source address match those at the destination address. If verification passes, then command returns success (CMDSTAT=PASS); else fail (CMDSTAT=FAIL).

EXAMPLES:

- cm -2 0x400000 0x402000 0x400
  Copy 1024 bytes of data from 0x400000 to 0x402000 using a memory access width of 2 bytes.

NOTES:

- The command assumes that the memory writes are simple. All memory written to is RAM. Attempts to write to flash or other non-volatile memory will fail. Be aware that this is true for all memory access commands except the "flash" command.

15.7 DHCP

Dynamic Host Configuration Protocol discover request.

USAGE:

dhcp -[brvV] [vsa]

DESCRIPTION:

The monitor implements a subset of the DHCP client protocol. Based on RFC2131 spec, the "automatic allocation" mode (DHCP server assigns a permanent IP address, or infinite lease, to a client) is the only mode supported. The DHCP handshake can be started in one of three different ways:

- Set the IPADD shell variable to "DHCP" in the monrc file.
- Issue this *dhcp* command from some other auto-bootable script.
- If no monrc exists and the DEFAULT_IPADD entry in config.h was set to DHCP when the monitor was built.

All DHCP startup mechanisms run the same client code; the only difference is that when DHCP is started through this command, there is no automatic retry (unless –r is specified) and no TFTP server interaction; it is left to the script to handle that. This is done to allow the user to override the automatic sequence of events discussed below with some other startup method. Aside from that difference, all other aspects of the handshake are identical (refer to discussion below).

Sequence of events for this limited implementation of DHCP...

- Client issues a DISCOVER: broadcast the fact that it is looking for a DHCP server.
- Server responds with an OFFER: server has received a DISCOVER request from the client. The offer may contain all the information that the DHCP client needs to boot-up, but this is dependent on the configuration of the server.
- Client issues a REQUEST: the client accepts the offer from the server and asks the server to send back the information it needs.
- Server responds with an ACK: reply from the server with the information requested.

Refer to discussion below for a more detailed discussion on monitor startup.

OPTIONS:

- -r
  Enable retries when invoked at the command line.
- -v
  Limited verbosity enabled;
- -V
  Extreme verbosity enabled;
- -b
  Run BOOTP;

NOTES:

- The use of various shell variables mentioned in this discussion provides an application with a lot of flexibility. It is beyond the purpose of this client to support all possible DHCP configurations, so some application-specific coding may be necessary. The monitor's DHCP code (dhcpboot.c & dhcp_00.c) is set up to allow application-specific code to replace dhcp_00.c without touching the generic DHCP client code in dhcpboot.c.
- The shell variable IPADD can also be set to DHCPV or DHCPv to enable different levels of verbosity during DHCP transactions... 'V' is full DHCP verbosity and 'v' only prints the DhcpSetEnv() calls (see common/monitor/dhcpboot.c).

- All of the above discussion applies to both BOOTP and DHCP. For BOOTP autostart, set the IPADD shell variable to BOOTP instead of DHCP (v & V extensions apply).

SHELL VARIABLES CREATED and/or USED BY DHCP:

Refer to Chapter 14 for details
DCLIPORT, DHCPDONTBOOT, DHCPCLASSID, DHCPCLIENTID, DHCPFLAGS, DHCPRELEASETIME, DHCPOFFRFLTR, DHCPRETRYTUNE, DHCPRQSTLIST, DHCPSTARTUPDELAY, DHCPVSA, DSRVRPORT

15.8 DIS

Disassemble memory.

USAGE:

dis –[m] {address} [linecount]

DESCRIPTION:

Display memory as instruction mnemonics for the target CPU. The format of this output is CPU specific. Depending on the port and the configuration of the monitor, this disassembly may include symbolic references to function calls.

OPTIONS:

- -m
  The 'more' option just runs the disassembler in a user-interactive mode allowing the user to just hit the spacebar to disassemble another line or block of lines.

15.9 DM

Display memory.

USAGE:

dm -[24bdefl:msv:] {address} {count}

DESCRIPTION:

There are several different types of memory that can reside on a target.

- **Standard memory:** used by the CPU for instruction and/or data storage. This is addressed sequentially with the options to specify the width of the memory. The width specification affects both the display format as well as the access type (byte, short, and long) used to retrieve the data.
- **FIFO memory:** can be read/written to empty/fill a FIFO. This type of access is not sequential, the same address is used for each access and the FIFO itself handles the queuing or de-queuing of the data. This is also width dependent.
- **Memory mapped peripherals:** have their own unique "qualities" that put restrictions on how they can be accessed.

This command attempts to support all modes of memory access through different options. Width can be specified for 8/16/32 bit access; the exact number of accesses can also be specified. This is actually quite important for FIFO and/or memory mapped peripherals because you want to read the EXACT number of units of memory specified.

OPTIONS:

- -2
  Retrieve data as 16-bit (2 bytes) wide units.;
- -4
  Retrieve data as 32-bit (4 bytes) wide units;
- -b
  Dump memory in binary. This is typically used by a host-resident debugger that is connected to the target for faster data retrieval. Width can be specified for 8/16/32 bits.
- -d
  Display memory in decimal units instead of the default hex. Width can be specified for 8/16/32 bits.
- -e
  Display 2 and 4-byte memory units endian reversed.
- -f
  Access memory as a FIFO. The address is not incremented, and width can be specified for 8/16/32 bits.
- -l {len}
  Override the default size of each line DM is to output.
- -m
  Interactively provide a "more?" query to the user to support user-controlled throttling of the output.
- -s
  Display the data as a null-terminated ASCII string. Display continues through memory until a NULL is detected. The only width supported here is 8-bit.
- -v {varname}
  Load a single data element at location 'address' into the shell variable 'varname'. If -s option is also present, then the variable is loaded with the address after the NULL termination of the string just printed.

EXAMPLES:

- dm -2 0xa3000000
  Display in short (16-bit) format, a block of memory starting at location 0xa3000000.
- dm -2f 0xa0100000 32
  Pull 32 units out of a 16-bit FIFO residing at location 0xa0100000 in the memory map.

- dm -s 0xa0041000 32
  Display memory starting at address 0xa0041000 as if it were an ASCII string. If a null is detected prior to the 32nd byte, terminate; else terminate at the 32nd byte regardless.
- To print a list of concatenated, NULL-terminated strings...
  set NEXT 0x100000
  dm -v NEXT -s $NEXT
  dm -v NEXT -s $NEXT
  dm -v NEXT -s $NEXT
  etc...
- Each 'dm' command reloads the value of NEXT with the address after the string just printed.

NOTES:
- All command arguments in the monitor are assumed to be of the syntax requiring that hex data be entered with a leading 0x, octal with a leading 0 else assume decimal.

RUNTIME EXAMPLES:

```
uMON>dm 0x1c000
0001c000: 74 68 65 5f 72 61 69 6e   5f 69 6e 5f 73 70 61 69   the_rain_in_spai
0001c010: 6e 5f 69 73 5f 6a 75 73   74 5f 61 73 5f 77 65 74   n_is_just_as_wet
0001c020: 5f 61 73 5f 74 68 65 5f   72 61 69 6e 5f 69 6e 5f   _as_the_rain_in_
0001c030: 66 72 61 6e 63 65 00 00   00 00 00 00 00 00 00 00   france..........
0001c040: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c050: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c060: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c070: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................

uMON>dm -s 0x1c000
the_rain_in_spain_is_just_as_wet_as_the_rain_in_france

uMON>dm -d 0x1c000 4
0001c000: 116 104 101  95

uMON>dm -e2 0x1c000 2
0001c000: 6874 5f65

uMON>dm -2 0x1c000 2
0001c000: 7468 655f

uMON>dm -4 0x1c000 4
0001c000: 7468655f  7261696e  5f696e5f  73706169

uMON>dm -4e 0x1c000 4
0001c000: 5f656874  6e696172  5f6e695f  69617073
```

15.10 ECHO

Print a string to local terminal.

USAGE:

echo [arg1] [arg2] ...

DESCRIPTION:

This command prints its arguments (separated by blanks and terminated by newline) to the local terminal connection. If there are no arguments, a blank line is printed. A few backslash characters are accepted:

\b	backspace
\c	no newline
\n	newline
\r	carriage return
\t	tab
\x##	ASCII-coded hex
\\	backslash

RUNTIME EXAMPLES:

```
uMON>echo abcdefg
abcdefg

uMON>echo "abcd\nefgh\nijkl\nmnop"
abcd
efgh
ijkl
mnop

uMON>echo \x31\x32\x33
123

uMON>echo "to process whitespace, add quotes"
to process whitespace, add quotes

uMON>echo hey \t this is a backslash: \\
hey      this is a backslash: \
```

15.11 EDIT

Line-mode file editor for use with TFS.

USAGE:

edit -[b:c:f:i:m:rs:t] {filename}

DESCRIPTION:

This command allows the user to edit ASCII files that are stored in TFS. It is a simple line-based file editor that supports line insertion, deletion, display and search. If the file already exists in TFS, then the content of that file is copied to a buffer in RAM space and all interaction and modification with the content of the file is done in the buffer. It is not until the 'q' (quit) command is issued, that the file is written to flash. If at any point during the edit session, the 'x' (exit) command is issued, then there is no change to the original file.

OPTIONS:

- -b {addr}
  Specify the buffer address that edit is to use for temporary storage of the file while being edited; if not specified, then edit assumes it owns all RAM on the board and the buffer starts at the address specified in the APPRAMBASE shell variable.;
- -c {cmd}
  In-line command executed prior to entering interactive mode;
- -f {flags}
  Flags that are applied to the newly created file (see TFS for flag description);
- -i {info}
  Information field applied to the newly created file (see TFS);
- -m {size}
  Use monitor's malloc to allocate buffer space
- -r
  By default, edit will automatically remove carriage returns (DOS style) and insert a final linefeed (if one is not present) at the end of the file being edited. This shuts that automatic stuff off.
- -s {size}
  Size of buffer to use for temporary storage.
- -t
  Convert tabs to spaces.

NOTES:

- A typical usage of the edit command will put the user in an interactive mode that supports a basic set of editing commands. At the startup of the interactive mode, edit will display the address of the buffer it will be using for temporary storage, followed by the message "*type ? for help".* Following is a list of commands supported for interactive mode...

d{LRNG}	delete line specified by "LRNG" (see below)
e#	edit line # (uses the same line editor as is used by the command line editor)
i	begin "insert" mode (use '.' to exit insert mode)
a	begin "append" mode (use '.' to exit append mode)
P[LRNG]	print entire buffer with line numbers prepended
p	print entire buffer
q[fname]	quit edit, write file (with no fname specified) it writes the to file originally opened
s[srchstr]	go to next line that contains "srchstr"
x	exit edit, do not write file
#	go to line # (use '$' to go to last line)
+/-#	go to line relative to current position

Where...
represents a decimal number;
LRNG represents a line number or inclusive line range (# or #-#);

Let's work through a simple example...
Assume we have the file 'monrc'. We want to change a line in monrc. We will simply delete the line and re-enter the new line (there are other ways, but lets keep the example simple). For the sake of this example, the response to each command will be highlighted, comments will be underlined and commands entered in bold. The output of "tfs cat monrc" is as follows...

```
uMON> tfs cat monrc                    Display the current content of the file.
set ETHERSPEED 10
set PROMPT "maint "
set MONFLAGS nophdr
set GIPADD 135.17.115.1
set IPADD 135.17.115.215
set NETMASK 255.255.255.0
uMON> edit monrc                       Start the editing process.
Edit buffer = 0xf000c000.
Type '?' for help
P                                      Print entire file with line numbers prepended.
  1: set ETHERSPEED 10
  2: set PROMPT "maint "
  3: set MONFLAGS nophdr
  4: set GIPADD 135.17.115.1
  5: set IPADD 135.17.115.215
  6: set NETMASK 255.255.255.0
d4                                     Delete line number 4
P                                      Print and notice the removal of the old line 4...
  1: set ETHERSPEED 10
  2: set PROMPT "maint "
  3: set MONFLAGS nophdr
  4: set IPADD 135.17.115.215
  5: set NETMASK 255.255.255.0
3                                      Move "current position" pointer to line 3
P                                      Print and notice the pointer moved to line 3...
  1: set ETHERSPEED 10
  2: set PROMPT "maint "
  3: set MONFLAGS nophdr
  4: set IPADD 135.17.115.215
  5: set NETMASK 255.255.255.0
a                                      Go into append mode (after current line pointer)
set GIPADD 135.17.115.2                <-- This is the text being added
.                                      Terminate append mode
P                                      Print and notice new line 4 and pointer at line
                                       5...
  1: set ETHERSPEED 10
  2: set PROMPT "maint "
  3: set MONFLAGS nophdr
  4: set GIPADD 135.17.115.2
  5: set IPADD 135.17.115.215
  6: set NETMASK 255.255.255.0
q                                      Write the new file to TFS and quit the editor
uMON> tfs cat monrc                    Display the new file with the modification...
set ETHERSPEED 10
set PROMPT "maint "
set MONFLAGS nophdr
set GIPADD 135.17.115.2
set IPADD 135.17.115.215
set NETMASK 255.255.255.0
uMON>                                  All done!
```

15.12 ETHER

Ethernet driver interface

USAGE:

ether -[d:p:tv:V] {on | off | stat | {print I|i|O pktlen}}

DESCRIPTION:

When the monitor starts up, the Ethernet interface is enabled. This command allows the user to display current statistics (errors, packet counts, etc), optionally turn the interface off and/or on, adjust verbosity of various portions of the Ethernet interface and use the Ethernet driver's code to descriptively print a packet in memory.

OPTIONS:

- -d {0|1}
  1=driver debug mode enabled;
- -p {0|1}
  1=accept all packets (promiscuous mode);
- -v {vflags}
  enable specific verbosity:
  - 0 turn off verbosity
  - a enable ARP trace
  - c print checksum error message
  - C dump checksum error packet
  - d enable DHCP trace
  - g enable GDB trace
  - i incoming packets (minus broadcast)
  - I incoming packets (including broadcast)
  - o outgoing packets
  - p phy r/w accesses
  - t enable TFTP packet trace
  - x hex dump (requires i, I or o)
  - X same as 'x' but with ASCII
- -V
  enable all verbosity (same as -vIodx)

EXAMPLES:

- ether -vio on
  Enable the Ethernet interface with verbosity on for incoming and outgoing packets.
- ether stat
  Dump the current statistics of the active Ethernet driver (packet counts, errors, etc…)
- ether print I 0x1234 120
  Assuming the 120 bytes of data starting at location 0x1234 in memory is the content of some incoming packet, this command will attempt to intelligently print the content in a verbose form.

15.13 EXIT

Terminate a script from any point within that script.

USAGE:

exit -[e:r]

DESCRIPTION:

This provides a clean and simple way to terminate a script from anywhere within the script. If the -r option is specified, then the script will automatically be deleted after exit is complete and the file has been closed under TFS. This is useful for a script that needs to be run only once, it can be placed in TFS as an autobootable script, then after a reset it will do its thing and delete.

OPTIONS:

- -e {executable}
  Launch some other executable immeidately after the running script exits. This allows the script to terminate so that the executable that it launches can then remove (or replace) it if necessary. Otherwise, if an executable program is launched from a running script, that script cannot be deleted by that application because TFS sees it as active.
- -r
  After exiting from the script, the file is automatically removed from TFS.

EXAMPLE:

- Within a script file:

```
# TOPofScript
set CNT 0

# NEXT:
gosub ECHO_CNT
if CNT lt 10 goto NEXT
exit

# ECHO_CNT:
echo $CNT
set -i CNT
sleep 1
return
```

15.14 FATFS

FAT File System Operations.

USAGE:

fatfs [options] {operation} [args]

DESCRIPTION:

This command supports the ability to overlay the FAT file system format on top of some block-readable device. It supports the following high-level capabilities…

- list files within specified directories
- dump the content of ASCII file
- transfer file to FAT from memory or TFS
- transfer file from FAT to memory or TFS
- query for the presence of a file within FAT space
- remove a file from FAT space

This command can be used with any interface that provides the ability to read and write blocks. It just needs to know the address of the readblock() and writeblock() functions to use. By default the read and write function pointers are assumed to be stored in the shell variables FATFS_RD and FATFS_WR respsectively. This command line level function pointer linkage allows fatfs to hook to any device at runtime by simply re-establishing the access pointers.

In all cases, this implies that a lower level interface must accompany this command. The most common is likely to be compact flash; hence, the 'cf' command could be used to initialize the compact flash interface and assign its access functions to FATFS. Refer to that command for details; however, note that this is independent of the interface; hence, it could be an SPI interface, SD, MMC, USB or whatever. As long as the interface command (be it "spi", "sd", "mmc", "usb", "cf" or whatever) has the ability to store the pointers to its read/write block-access functions in the FATFS shell variables.

OPTIONS:

- -r {addr}
  set the pointer to be used as the read-block access function (default: FATFS_RD);
- -s {addr}
  base address of RAM scratch space used by fatfs (default: APPRAMBASE);
- -w {addr}
  set the pointer to be used as the write-block access function (default: FATFS_WR);
- -v
  additive verbosity;

OPERATIONS:

- cat {filename}
  Dump the content of the specified ASCII file to the console.
- get {fat_file} {tfs_file | address}
  Transfer the content of a FAT file to either a TFS file or directly to memory. To distinguish whether or not the destination is a TFS file or addres, the command assumes that a "0x" prefix will be used to specify the hex address.
- put {tfs_file | address,size} {fat_file}
  Similar to "get", but now the opposite direction… Transfer the content of a TFS file or block of memory to a FAT file. To distinguish whether or not the destination is a TFS file or addres, the command assumes that a "0x" prefix will be used to specify the hex address, plus, the size is assumed to be a comma-delimited number following the addres (with no whitespace separation).
- ls [dirname]
  If no argument is specified, this command will list the highest level directory on the device that FATFS is connected to.

- qry {fat_file_name_filter}
  Query for the presence of a file in the FAT space. If the file is found, the variable FATFSTOT will be loaded with some positive number (else zero). Also, the shell variable FATFSNAME and FATFSSIZE will be loaded with the name and size (respectively) of the last filter match. The filtering capability is basic. An initial or trailing (or both) asterisk is used to support some basic filtering.
- rm [fat_file]
  Remove specified file from FAT space.

LIMITATIONS:

As of this writing, the removal and/or creation of a directory is not supported:

CREDITS:

FAT filesystem code based on the free DOSFS library by Lewin Edwards, available from http://www.zws.com/products/dosfs/. This code was first incorporated into MicroMonitor by Graham Henderson.

EXAMPLE USAGE:

```
uMON>cf init FATFS
uMON>fatfs ls
Volume:     UMON_CF
09/20/2006  11:44   <DIR>            TMP
09/20/2006  11:44             12288  BASHRC~1.SWP    A
09/20/2006  11:44             35331  3DCAL_~1        A
09/20/2006  11:44               209  RUSH.HST        A
09/20/2006  11:44              6450  SDATLOG.TXT     A
09/20/2006  11:44               270  UCONLO~1        A
uMON>fatfs ls tmp
09/20/2006  11:44   <DIR>            .
00/17/2006  01:01               473  MONRC
uMON>fatfs cat TMP/MONRC
echo "this is a monrc file"
set IPADD DHCPv
uMON>
```

15.15 FLASH

Flash memory operations.

USAGE:

flash {operation} [args]

DESCRIPTION:

Flash is writeable non-volatile memory. This command allows the user to operate on flash without being aware of the handshaking necessary to erase, write and configure the flash devices on the target. The code that makes up the flash command in the monitor supports multiple banks of flash, and also allows each bank to be a different bus width. This, of course, is dependent on the underlying hardware. Several flash operations are supported, some of which depend on the capabilities of the underlying flash device…

- "bank [#]"
  If no argument is specified, this command returns the currently active bank; if an argument is provided, then that value becomes the new default bank. Note that the number of banks supported is hardware-platform dependent. Note that as of uMon1.0, the need to know the bank number is being slowly (but surely) eliminated.
- "erase {sector # | all}"
  Allows the user to erase a specified sector of the flash or all of the non-protected sectors (all). The erasure brings all bits to 1; hence the data would be read as 0xffs.
- "ewrite {dest} {src} {bytecnt}"
  This command allows the user (typically only the owner of the monitor code) to re-write the protected sectors with a new code. This essentially allows the monitor to re-write itself. The arguments are a destination address where the data is to be written in flash, the source address of the data to be placed in flash and the number of bytes to be written.
- "init"
  Initializes all internal flash operation data structures (typically only needed for monitor development).
- "info"
  Queries each bank configured in the hardware and displays each sector's address range, erased, locked and protected state. Plus, as of uMon1.0, it also displays whether or not the sector is owned by TFS. This command will also populate shell variables with the results of the query just made: FLASH_BASE_N (base address of device 'N', FLASH_SCNT_N (total number of sectors in device 'N', FLASH_END_N (end address of device 'N' & FLASH_DEVTOT (total number of devices).
- "lock"
  If the device supports it, then this command will lock the specified sector(s) so that future writes will fail. If no sector or range of sectors is specified, then the sectors owned by the uMon executable image are locked[89].
- "unlock"
  If the device supports it, then this command will unlock the specified sector(s) so that future writes will be legal. Similar to the "lock" command, if no sector or range of sectors is specified, then the sectors owned by the uMon executable image are locked.
- "lockdwn"
  Some devices support the ability to lock a sector, then only through a power-cycle, reset or assertion of a hardware-specific pin on the chip, can they be unlocked. This facility is avaible in the flash command; however, isn't implemented on very many targets. It doesn't have a corresponding "unlockdwn" because that is typically harwdare dependent and requires something that, if implemented properly, should not be accessible by firmware.
- "opw"
  Enables a flag that allows the very next command to the monitor to be a flash operation that will ignore the protected state of the flash. By default the monitor knows what parts of the flash must be protected from random writes (the FLASH_PROTECT_RANGE definition in config.h determines this at monitor build

[89] Actually, the sectors that are locked here are those specified by the FLASH_PROTECT_RANGE definition in the monitor's config.h file. By definition, this range specification should at a minimum include the sectors used by the uMon image.

time). This command provides the user with a "1-command-window" to override that protection. Must be in user-level 3 mode to do this.

- "prot"
  This sets a flag that is used in the flash command to provide "soft" protection.
- "unprot"
  This clears the flag that is used in the flash command for "soft" protection.
- "write {dest} {src} {bytecnt}"
  This command allows the user write to the flash. Like the ewrite command, it assumes that the data to be written is at the location specified by "src" and the number of bytes is specified by "bytecnt".

PROT vs LOCK vs LOCKDWN…

The flash command has three different mechanisms for protecting against an illegal (or unwanted) write to flash space: protect, lock and lock down. Each mechanism has its place, and in the nutshell, it just depends on what capability comes with the flash device on the hardware.

The flash "prot" protection can be applied to any device. It doesn't require any specific capability built into the flash. Nowadays, the majority of flash devices used have some mechanism for hardware-assisted protection; however, a few years back that wasn't the case; hence the soft protection was handy. If a sector is protected by the "prot" operation, then by default, the "write/ewrite/erase" operations will be rejected at the command line. The only way to modify a sector protected this way is to issue the "opw" (open protection window) operation. Then, the very next command is the opened window; hence, the very next command can be a write/ewrite/erase.

The flash "lock" protection can be used with devices that have the ability to issue a "lock" command to a sector, thus requiring the device to have a device-specific handshake applied to it (i.e. unlock) prior to being able to modify the content of that sector.

Finally, **the flash "lockdwn" protection** can be used with devices that have the ability to issue a "lockdwn" command to a sector. In this mode (for devices that support it), the flash is then locked (i.e. unmodifiable) until it is either reset or power-cycled, or a specific pin on the device is asserted.

Each of the above have their purpose and value. Each have different requirements put on the underlying flash device and the hardware. The "prot" protection doesn't require any specific hardware support; however, it provides the weakest protection. Sooner or later, all devices are likely to support some hardware-assisted lock mechanism; hence, sooner or later uMon will probably eliminate the "prot" protection method.

FM

Fill a memory range with a specified value.

USAGE:

fm -[24cnip] {source address} {destination address | count} {value | pattern}

DESCRIPTION:

Fill a block of memory with a specified value or pattern.

OPTIONS:

- -2
  Access data as 16-bit (2 bytes) wide units.;
- -4
  Access data as 32-bit (4 bytes) wide units;
- -c
  Arg2 is count, in bytes.
- -i
  Increment the value loaded into memory.
- -n
  Disable the verify used with standard 1,2&4-byte width accesses.
- -p
  Arg3 is a pattern, applicable only with 8-bit access width.

EXAMPLES:

- fm -2 0x400000 0x402000 0x400
  Fill memory range from 0x400000 through 0x402000 with 0x400. Access as 16-bit memory.
- fm -c 0x400000 0x2000 0x99
  Fill 0x2000 bytes of memory starting at 0x400000 with 0x99.

RUNTIME EXAMPLES:

```
uMON>fm -c $APPRAMBASE 64 0xff
uMON>dm $APPRAMBASE 64
0001c000: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
0001c010: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
0001c020: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
0001c030: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
uMON>fm -cp $APPRAMBASE 64 12345678ABCDEFGH
uMON>dm $APPRAMBASE 64
0001c000: 31 32 33 34 35 36 37 38   41 42 43 44 45 46 47 48   12345678ABCDEFGH
0001c010: 31 32 33 34 35 36 37 38   41 42 43 44 45 46 47 48   12345678ABCDEFGH
0001c020: 31 32 33 34 35 36 37 38   41 42 43 44 45 46 47 48   12345678ABCDEFGH
0001c030: 31 32 33 34 35 36 37 38   41 42 43 44 45 46 47 48   12345678ABCDEFGH
uMON>fm -ic $APPRAMBASE 64 0
uMON>dm $APPRAMBASE 64
08200000: 00 01 02 03 04 05 06 07   08 09 0a 0b 0c 0d 0e 0f   ................
08200010: 10 11 12 13 14 15 16 17   18 19 1a 1b 1c 1d 1e 1f   ................
08200020: 20 21 22 23 24 25 26 27   28 29 2a 2b 2c 2d 2e 2f    !"#$%&'()*+,-./
08200030: 30 31 32 33 34 35 36 37   38 39 3a 3b 3c 3d 3e 3f   0123456789:;<=>?
uMON>
```

15.16 GOSUB

Call a subroutine within the current script.

USAGE:

gosub {tag}

DESCRIPTION:

As an extension to the goto capability within TFS scripts, this command branches to the named tag, and assumes that at some point the code branched to will execute a return. At that point, the script will continue execution on the line after the gosub line.

EXAMPLE:

- Within a script file:

```
# TOPofScript
set CNT 0

# NEXT:
gosub ECHO_CNT
goto NEXT

# ECHO_CNT:
echo $CNT
set -i CNT
sleep 1
return
```

15.17 GOTO

Branch to a tagged line in a TFS script.

USAGE:

goto {tag}

DESCRIPTION:

Since TFS supports executable files that can be a set of monitor commands, this command enhances that capability by allowing the script to branch to specific tags within the script. The tag is simply a line starting with a # (pound sign), one blank and a tag. For example, # *TAG* on a line by itself is a target that could be branched to by the command *goto TAG*.

EXAMPLE:

- Within a script file:

```
# TOPofScript
set CNT 0

# ECHO_CNT
echo $CNT
set -i CNT
sleep 1
goto ECHO_CNT
```

15.18 HEAP

Display/modify current heap statistics.

USAGE:

heap -[cf:m:vX:x]

DESCRIPTION:

This command allows the user to peek into the internal structures allocated by the monitor's memory allocator. With no options specified, a summary of the heap statistics is printed; add -v to dump all structures within the heap.

OPTIONS:

- -c
  Clear high-water level and malloc/free totals
- -f {pointer}
  Call "free" to release the block of memory pointed to by "pointer";
- -m {size}
  Call "malloc" to allocate a block of memory of size "size";
  This will load the shell variable MALLOC with the result.
- -v
  Verbose mode
- -X {start,size}
  Extend the monitor's heap using "size" bytes beginning at "start";
- -x
  Disable the monitor's extended heap (if not in use)

EXAMPLES:

- heap
  Dump heap summary, i.e...

```
Heap summary:
  Malloc/free calls:  106/103 (delta=3)
  Malloc/free totals: 270096/270040
  High-water level:   46876
  Malloc failures:    0
  Bytes overhead:     1020
  Bytes currently allocated:   644
  Bytes free on current heap:  50856
  Bytes left in allocation pool:  479960
```

- heap -v
  Verbose listing of heap statistics, i.e...

```
       addr     size free?    mptr        nxt        prv      ascii@addr
  0: 0x0000288c    12   n   0x00002878 0x00002898 0x00000000  ......(.....
  1: 0x000028ac    36   n   0x00002898 0x000028d0 0x00002878  ENTRYPOINT......
  2: 0x000028e4    12   n   0x000028d0 0x000028f0 0x00002898  ..) ..)...)<
  3: 0x00002904     8   n   0x000028f0 0x0000290c 0x000028d0  PROMPT..
  4: 0x00002920     8   n   0x0000290c 0x00002928 0x000028f0  uMON...
  5: 0x0000293c    12   n   0x00002928 0x00002948 0x0000290c  ..)|..)\..).
  6: 0x0000295c    12   n   0x00002948 0x00002968 0x00002928  APPRAMBASE..
  7: 0x0000297c     8   n   0x00002968 0x00002984 0x00002948  0xf000..
  8: 0x00002998    12   n   0x00002984 0x000029a4 0x00002968  ..)...)...).

...


 45: 0x00002e24    16   n   0x00002e10 0x00002e34 0x00002df0  BOOT_LINE_ADRS..
 46: 0x00002e48     8   n   0x00002e34 0x00002e50 0x00002e10  0x17000.
```

```
47: 0x00002e64    80   n   0x00002e50 0x00002eb4 0x00002e34  ..D ..=P..E...F.
48: 0x00002ec8     8   n   0x00002eb4 0x00002ed0 0x00002e50  0x18000.
49: 0x00002ee4  6548   y   0x00002ed0 0x00200000 0x00002eb4  ......8...9x....
50: 0x00200014 44308   y   0x00200000 0x00000000 0x00002ed0  ................
 Malloc/free calls:  106/103 (delta=3)
 Malloc/free totals: 270096/270040
 High-water level:   46876
 Malloc failures:    0
 Bytes overhead:     1020
 Bytes currently allocated:   644
 Bytes free on current heap:  50856
 Bytes left in allocation pool:  479960
```

- heap -X 0x200000,0x80000
  Extend the monitor's heap by 0x80000 bytes using memory at location 0x200000.

NOTE:
After an extension is made to the amount of memory available for malloc (using -X); that extension can be "undone" with -x ONLY if there are no active allocations within the extension space.

15.19 HELP

Display command information.

USAGE:

help -[di] [command name]

DESCRIPTION:

Displays help text for a specific command (requires command name to be specified) or display a tabular listing of all commands available. Note that the string "help" or the character "?" will be seen by the monitor's command interpreter as the same command.

Note, if the “command name” argument is specified and “help” doesn’t find that command name within the built-in list, it will then check for the presence of that name as an executable file in TFS. If found, “help” will execute that command with a single argument “help”. For example, if there is an executable in TFS called “abc” and the user types “help abc”, the command will determine that “abc” is not a command in the built-in list and then determine that it is a TFS executable. At that point, help will internally issue the command: “abc help”.

OPTIONS:

- -d
  when displaying all commands available, a per-command description is included.
- -i
  dump monitor state (the old ‘mstat’ output)

RUNTIME EXAMPLES:

```
Micro-Monitor Command Set:
arp         call        cast        cm          dhcp        dis
dm          echo        edit        ether       exit        flash
fm          gdb         gosub       goto        heap        help
?           history     icmp        if          item        mt
mtrace      pm          prof        read        reg         reset
return      set         sleep       sm          strace      ulvl
tftp        tfs         unzip       xmodem

uMON>help -d
 arp          0 Address resolution protocol
 call         0 Call embedded function
 cast         0 Cast a structure definition across data in memory.
 cm           0 Copy Memory
 dhcp         0 Issue a DHCP discover
 dis          0 Disassemble binary
 dm           0 Display Memory
...
 return       0 Return from subroutine
 set          0 Shell variable operations
 sleep        0 Second or msec delay (not precise)
 sm           0 Search Memory
 strace       0 Stack trace
 ulvl         0 Display or modify current user level.
 tftp         0 Trivial file transfer protocol
 tfs          0 Tiny File System Interface
 unzip        0 Decompress memory (or file) to some other block of memory.
 xmodem       0 Xmodem file transfer
 version      0 Version information
```

15.20 HISTORY

Display history of most recent commands issued at monitor command line.

USAGE:

history

DESCRIPTION:

The monitor supports command line editing capabilities. Depending on what was configured at build time, the monitor supports a subset of the vi-like commands sequences used in KSH & BASH. Or, if configured to use the VT100 editing, then the arrow keys are used. In either case, some number of previously issued commands are accessible from the command line with ESC-k or the UP-ARROW to step back through the command history and ESC-j or the DOWN-ARROW to step forward through the command history. The standalone "*history*" command will dump the most recently executed commands.

RUNTIME EXAMPLE:

```
uMON>echo abc
abc

uMON>echo def
def

uMON>echo ghi
ghi

uMON>history
echo abc
echo def
echo ghi
history
uMON>
```

15.21 ICMP

ICMP operation support.

USAGE:

icmp -[c:d:f:rv:] {operation} [args]

DESCRIPTION:

Allows the user to run specific ICMP-based queries on the local LAN. Currently, the ICMP TIMESTAMP and ECHO are the only supported operations.

OPTIONS:

- -c {##}
  repetition count for echo;
- -d {##}
  delta (in hours) relative to GMT;
- -f {x|d}
  hex or decimal output (default is ASCII);
- -r
  check server resolution;
- v {varname}
  load result into shell var "varname" (see note below).

OPERATIONS:

- time
  Issue ICMP TIMESTAMP request.
- echo
  Basic ping capability.

EXAMPLES:

- icmp time 135.3.29.35
  Issue an ICMP time request to server on 135.3.29.35.
- icmp echo 135.3.29.39
  Similar to ping 135.3.29.39.

NOTES:

- The *d,f,r & v* options are for the time operation and *c* is for echo.
- If the –v option is used, then the result of the icmp transaction will be loaded to the shell variable specified as the argument to the –v option. For the 'icmp time' command, the format of the result depends on the use of the –f option (hex, decimal or ASCII). It is stored as a number (hex or decimal) or a time string (HH:MM:SS.MSEC). Then, if the time is non-standard, the content of the variable is appended with ".ns". For the 'icmp echo' command, if the echo succeeds, then the variable is loaded with ALIVE, else the variable is loaded with NOANSWER.

RUNTIME EXAMPLE:

```
uMON>icmp echo $GIPADD
192.168.1.1 is alive
```

15.22 IF

Conditional test with branching.

USAGE:

if -[t:v] [{arg1} {compare} {arg2}] {action} else {action}

DESCRIPTION:

To support scripting capability through TFS files, this command allows the user to build conditional tests that result in branches to different points within a executable script. The most common use of this is to make a string or numerical comparison between two different arguments. If the -t option is specified, then that overrides the basic comparison and the argument to -t is considered to be the test.

Numerical comparisons:

- gt
  true if arg1 is greater than arg2
- lt
  true if arg1 is less than arg2
- le
  true if arg1 is less than or equal to arg2
- ge
  true if arg1 is greater than or equal to arg2
- eq
  true if arg1 is equal to arg2
- ne
  true if arg1 is not equal to arg2

Logical comparisons:

- and
  true if arg1 AND arg2 is non-zero
- or
  true if arg1 OR arg2 is non-zero
- xor
  true if arg1 XOR arg2 is non-zero

String comparisons:

- seq
  true if the string of arg1 is identical to the string of arg2
- sec
  true if the string of arg1 is identical to the string of arg2 ignoring case
- sne
  true if the string of arg1 is not identical to the string of arg2
- sin
  true if the string of arg1 is within the string of arg2

Actions:

- goto tag
  Jump to the location in the script specified by "tag".
- gosub tag
  Call the subroutine specified by "tag"
- return
  Return from the currently active subroutine.
- exit
  Terminate the currently active script.

OPTIONS:

- -t {testtype}
  override the default "arg1 compare arg2" comparison;
  testtypes:

gc	"gotchar"... a character is present on the UART interface
ngc	"not-gotchar"... a character is not present on the UART interface
iscmp {filename}	"is-compressed"... specified file is compressed

- -v
  verbose mode. Simply prints “TRUE” or “FALSE” after the test or comparison.

EXAMPLES:

- if $VAR1 seq $STR1 goto MATCH
  If the string contained within $VAR1 is the same as the string contained in $STR1 then branch to the line containing the tag 'MATCH'.
- if -t gc goto GOT_ONE
  If there is a character pending on the UART interface, branch to the line containing the tag 'GOT_ONE'.
- if -t iscmp app_file gosub COMPRESSED
  If the file "app_file" is compressed, then call the subroutine 'COMPRESSED'.
- if $VAR seq \$VAR goto VAR_NOT_SET
  If the shell variable 'VAR' does not exist, branch to the subroutine 'VAR_NOT_SET'

15.23 ITEM

Process a list of strings.

USAGE:

item {idx} {stor_var} [item1] [item2] [item3] .....

DESCRIPTION:

This command, in conjunction with "if" and "goto", allows the user to build scripts that conveniently process a list of strings (or items). Consider the 3rd through Nth arguments (itemN above) to be a list or table of items. The first argument (idx) is used as the index into this list (starting with 1) and the second argument is the name of the shell variable into which the item is to be placed. If the index is out of range, the variable is cleared.

EXAMPLES:

- item 2 letter a b c d e f
  This would place 'b' in the shell variable "letter"
- item $idx letter a b c d e f g
  If the content of $idx was between 1 and 7, then the shell variable "letter" would contain one of the letters a through g respectively. For any other value in $idx, the shell variable "letter" would be empty.

15.24 JFFS2

Journalling Flash File System-2 interface command

USAGE:

jffs2 [b:c] {operation} [operation-specific argument list]

DESCRIPTION:

For operating systems that need to allocate a portion of the flash memory to JFFS2, this command provides access to files in JFFS2 space and allows the user to copy those files from JFFS2 to TFS or memory. Similar in basic format to uMon's TFS command, there are several operations within the JFFS2 command that support listing files, copying a specified file from JFFS2 to TFS (or raw memory), querying for the presence of a particular file within JFFS2 space, and also several different mechanisms for displaying data within the JFFS2 nodes themselves.

The command assumes that some portion of the memory is allocated to the JFFS2-formatted flash file system. This block of flash space is identified by the JFFS2 command in one of several different ways. First the command looks for the JFFS2BASE shell variable and if found will use the content of that variable as the base address. Second, there's the –b {address} option. If this option is present the address specified as the argument to the option overrides the presence of both JFFS2BASE. If none of the methods mentioned are present, then the base address of the JFFS2 flash space just defaults to the DEFAULT_JFFS2_BASE definition established in config.h (or zero or not set in config.h).

The jffs2 command can be included in the uMon build as a built-in or it can be built as a TFS-based application. The makefile for doing this comes with the uMon tarball and is found under umon/umon_apps/jffs2. The makefile is similar to the makefile used with the other application demonstrations discussed in this user manual. Also under that directory is an example image file for big-endian (jffs2_be.bin) and little-endian (jffs2_le.bin). These files can be transferred to target memory; thus, allowing a user with a board that doesn't have the jffs2 command to try it out prior to building it in to the uMon internal command table. Plus it provides a working example of a JFFS2 formatted flash area.

JFFS2 operations:

- **cat** {filename}
  Dump the content of the specified file (assumed to be ASCII) to the console.
- **cp** {JFFS2 filename} {TFS filename | hex address}
  Copy the content of a JFFS2 file to either a TFS file or just to raw memory.
- **dirent** *
  List all 'dirent' type nodes sequentially as they are found in the JFFS2 space.
- **dump** *
  List all nodes sequentially as they are found in the JFFS2 space.
- **ino** [range] *
  List all nodes associated with a specified inode or range of inodes.
- **ls** [filter]
  List files and/or directories hierarchically. If a filter is specified, then list only those that match the filter. The filter is specified in one of four types: sss, *sss, sss* or *sss* (where 'sss' is any string). This operation will load the shell variable JFFS2NAME with the most recent filter match, and JFFS2SIZE with the size of that file. It also loads JFFS2TOT with the number of matches.
- **qry** [filter]
  This command is essentially the same as 'ls' above, except that there is no output to the console. The results are simply loaded into the shell variables as specified above.
- **node** [range] *
  List range of nodes within JFFS2 space.
- **ntot ***
  Display total number of nodes in JFFS2 space.
- **pino** {inode #} *
  List nodes with the specified parent inode.

- **zinf** {source address} {destination address} {source length} {destination length}
  Run the zlib inflator that comes with JFFS2 on the specified block of memory.

* The commands marked with the asterisk are primarily there for querying the raw content of the JFFS2 formatted space, and were added primarily for debug and development of the JFFS2 command within uMon. It is not recommended that these commands be used in a deployed script simply because they may be removed in the future.

OPTIONS:

- -b {address}
  Specify the base address of the JFFS2 space. This can also be retrieved from the shell variable JFFS2BASE.
- -c
  By default the CRC checks are disabled (for speed). This option enables all crc checks on name, header and data.

NOTES:

- This command is useful for booting Linux. It can be used in several different ways to aid in kernel startup. For example, it can be used to look for the presence of a file to determine how to boot up. It can look for the presence of a file and then copy it to TFS and use it as a startup script. This allows the linux application to establish its own bootup strategy because the bootscript eventually run by uMon out of TFS can be created by a Linux application, then on next bootup, uMon can look for that file in JFFS2, copy it to TFS and run it.
- The JFFS2 command uses some scratch memory space, which by default it assumes is at the address specified by the content of the APPRAMBASE shell variable. If this space is being used for something else, then the APPRAMBASE shell variable should be changed to point to the space to be used by JFFS2; then, upon completion of JFFS2 command, set it back to it's original value.

15.25 MT

Run memory diagnostics

USAGE:

mt -[cqs:S:t:v] {address} {size}

DESCRIPTION:

Runs walking ones and address-on-address test across the memory range specified.

OPTIONS:

- -c
  Run continuously
- -C
  Run crc32 on a specified block of memory. Load shell variable MTCRC with the result.
- -q
  Quit at error;
- -S
  Attempt to determine the size of the memory space. With this option, 'address' is assumed to be the base of physical RAM space and 'size' is the maximum possible size of RAM.
- -s ##
  Sleep ## seconds between the address-in-address read and write.
- -t ##
  For the address-in-address test, toggle the data every ##-bit block, where '##' can be 32 or 64
- -v
  Verbose mode;

EXAMPLES:

- mt 0x300000 0x100
  Test 256 bytes of memory starting at 0x300000. The address-in-address is untouched, so the output of "dm" for this test is…

```
uMON>dm 0x300000 64
00300000: 00 30 00 00 00 30 00 04   00 30 00 08 00 30 00 0c   .0...0...0...0..
00300010: 00 30 00 10 00 30 00 14   00 30 00 18 00 30 00 1c   .0...0...0...0..
00300020: 00 30 00 20 00 30 00 24   00 30 00 28 00 30 00 2c   .0. .0.$.0.(.0.,
00300030: 00 30 00 30 00 30 00 34   00 30 00 38 00 30 00 3c   .0.0.0.4.0.8.0.<
```

- mt –t32 0x300000 0x100
  Test 256 bytes of memory starting at 0x300000, but invert every 32-bit block for the address-in-address test. The output of "dm" for this test is…

```
uMON>dm 0x300000 64
00300000: 00 30 00 00 ff cf ff fb   00 30 00 08 ff cf ff f3   .0.......0......
00300010: 00 30 00 10 ff cf ff eb   00 30 00 18 ff cf ff e3   .0.......0......
00300020: 00 30 00 20 ff cf ff db   00 30 00 28 ff cf ff d3   .0. .....0.(....
00300030: 00 30 00 30 ff cf ff cb   00 30 00 38 ff cf ff c3   .0.0.....0.8....
```

- mt –t64 0x300000 0x100
  Test 256 bytes of memory starting at 0x300000, but invert every 64-bit block for the address-in-address test. The output of "dm" for this test is…

```
uMON>dm 0x300000 64
00300000: 00 30 00 00 00 30 00 04   ff cf ff f7 ff cf ff f3   .0...0..........
00300010: 00 30 00 10 00 30 00 14   ff cf ff e7 ff cf ff e3   .0...0..........
00300020: 00 30 00 20 00 30 00 24   ff cf ff d7 ff cf ff d3   .0. .0.$........
00300030: 00 30 00 30 00 30 00 34   ff cf ff c7 ff cf ff c3   .0.0.0.4........
```

15.26 MTRACE

Configure the monitor memory trace facility.

USAGE:

mtrace -[mn] {sub-command} [sub-command-specific arguments]

DESCRIPTION:

The memory trace facility within the monitor allows the developer to add "silent logging" to an application. This capability originated in the monitor as a result of a need to be able to debug XMODEM file transfer… Consider the situation: XMODEM is a simple protocol over a serial port that allows data to be transferred from target to host (or host to target). The problem is that while the protocol is active, the serial port cannot be used for any debug printout (assume only one serial port). To trace the XMODEM protocol in progress then, interaction had to be logged to memory and dumped after the protocol completed; hence the beginnings of mtrace.

This command is used in conjunction with the mon_memtrace() API function. The options allow the trace to be configured to a specific block of memory, trace can be temporarily disabled/re-enabled, cleared, or it can even be redirected to the console if the situation permits. A few immediate uses become apparent for mtrace:

- Following the above XMODEM scenario, the obvious first use is to provide a mechanism for debug logging when a serial port is not available.
- In interrupt handlers, where it is usually dangerous to insert printf(), this can be used and the output can be dumped later.
- As a general purpose system log that can be turned on and off as needed.

OPTIONS:

- -m
  Enable "more" output throttling to be used with the "dump" sub-command;
- -n
  Disable wrapping

COMMANDS:

- on
  If configured, then this just turns on the trace facility. If not turned on, then API calls to mon_memtrace() simply return. Note if "pron" was previously enabled, this transitions the trace mode back to memory only.
- off
  If configured, this turns off the trace facility (mon_memtrace() just returns).
- dump
  Dump all lines of trace that have accumulated in the trace buffer. The buffer is a circular queue, so wrap may occur. Note that this does not clear the queue (use reset for that).
- pron
  The trace is logged to memory (as usual), but is also sent to the console.
- reset
  Clear all accumulated trace, but maintain previously established configuration.
- log {msg}
  Manually add an entry to the trace log.
- mip {base}
  Reset the base address of the trace buffer. Typically this would be used after a trace session was active, but the target reset. If the trace buffer is not corrupted, the pointers can be reestablished and the trace buffer can be dumped.
- cfg [{base} {size}]
  Configure the trace buffer base address and size. If no arguments are given, then the current configuration is displayed.

15.27 PM

Put memory local to the target.

USAGE:

pm -[24aefoSsx] {address} {value | string} [value]

DESCRIPTION:

Refer to the *dm* command description (section 15.9) for a discussion on the different types of memory that can reside on a target. This command attempts to support all modes of memory access through different options. Width can be specified for 8/16/32 bit access, the exact number of accesses can also be specified. This is actually quite important for FIFO and/or memory mapped peripherals because you want to read the EXACT number of units of memory specified.

OPTIONS:

- -2
  Assume the width of the memory accesses is x2;
- -4
  Assume the width of the memory accesses is x4;
- -a
  Operation to perform is AND instead of a direct write.
- -e
  For data written as 2 and 4-byte units, endian-swap prior to placing into memory location;
- -f
  Assume the destination address is a FIFO;
- -o
  Operation to perform is OR instead of a direct write.
- -s
  Assume the data is an ASCII string and process \t\r\n appropriately. Populate the STRLEN shell variable with the size of the created string.
- -S
  Similar to -s except that the string is concatenated to the end of the string specified by the start address.
- -x
  Operation to perform is XOR instead of direct write.

EXAMPLES:

- pm -2 0xa3000000 0x4411
  Place in short (16-bit) format, 0x4411 at location 0xa3000000
- pm -f 0xa010000 0x32 0x33 0x34 0x35
  Assume a FIFO is memory mapped at address 0xa010000 and push those four bytes onto it.
- pm -s 0x401000 "hello world"
  Put the string "hello world" into memory at location 0x401000 and terminate it with a 0x00. The variable STRLEN would then be loaded with 11 (representing the length of the “hello world” string).
- pm -S 0x401000 " again"
  Concatenate the string that starts at address 0x401000 with the string " again".

15.28 PROF

Run-time profiler facility.

USAGE:

prof -[a:h:m:s:] {operation} [args]

DESCRIPTION:

Allows the user to configure the monitor's profiling facility for application-specific requirements. Refer to section 10.8 for implementation details. The profiler is a combination of monitor configuration and run-time data gathering. The prof command is used to configure the profiling prior to run-time data gathering and to dump the statistics gathered after the runtime profiling has completed.

OPTIONS:

- -a {##}
  address to use for statistics logging (default builds off of APPRAMBASE)
- -h {##}
  minimum hit-count used when statistics are displayed;
- -m {##}
  line count for output throttling (--more--) of displayed statistics;
- s {filename}
  file to be used for symbolic information (default is "symtbl")..

OPERATIONS:

- on
  Enable runtime profiling.
- off
  Disable runtime profiling.
- show
  Dump the statistics of a previously run profiling session.
- init
  Initialize all internal structures and tables.
- call {type} {pc} {tid}
  Call the profiler function specifying the type ('t', 'p' or 'f') the PC value and the TID value. The PC and/or TID value may be set to zero depending on the type of profiling request being made. 't' = TID, 'f' = FUNCTION, 'p' = PC
- restart
  Initialize all statistics gathered so that a configured profiling run can be restarted.
- tidcfg {tidtot}
  Initialize the TID profiler specifying the number of unique TIDs to be handled.
- funccfg
  Initialize the FUNCTION profiler from the symbol table file.
- pccfg {wid} {add} {size}
  Initialize the PC profiler with instruction width, base address of application's .text section and the size of this .text section.

15.29 READ

Interactive shell variable entry.

USAGE:

read -[fnp:t:T:] {varname1} [varname2] …

DESCRIPTION:

This command provides the monitor with the ability to run a script from TFS with the ability to interact with the user by requesting that the user input data that will be loaded into a specified shell variable.

OPTIONS:

- -f
  flush console input;
- -n
  do not echo characters as they are typed;
- -p {variable prefill}
  pre-load (and display) the content of the variable to be loaded
- -t {###}
  wait for input, but timeout after ### milliseconds, timeout is restarted after each character is received.
- -T {###}
  wait for input, but timeout after ### milliseconds, timeout is cumulative (doesn't restart after each character).

EXAMPLES:

- read name
  Wait forever for input from the user to load into the shell variable *name*. If the user responds with ENTER (no text) the shell variable is cleared; otherwise *name* is loaded with the first input token.
- read -t3000 name
  Wait for up to 3 seconds for the user to enter data into the shell variable *name*. If the timeout occurs, the shell variable *name* is untouched. If the timeout doesn't occur, but the user only types ENTER, the shell variable is cleared; otherwise *name* will contain the first token.

15.30 REG

Display or modify a value in the current monitor register cache.

USAGE:

reg [regname] [value]

DESCRIPTION:

Some ports of the monitor support debugging of an application. Upon hitting a breakpoint, the exception handler copies the CPU's context (registers) to a monitor-maintained array (or register cache). Then while the monitor is active the user has the ability to display and/or modify the value of any of these registers. It is important to note that this command does NOT affect the current value of the register; it is the register cache that is affected.

EXAMPLE:

```
uMON>reg
    PC=0x00000000     SR=0x00000000     A0=0x00000000     A1=0x00000000
    A2=0x00000000     A3=0x00000000     A4=0x00000000     A5=0x00000000
    A6=0x00000000     SP=0x00000000     D0=0x00000000     D1=0x00000000
    D2=0x00000000     D3=0x00000000     D4=0x00000000     D5=0x00000000
    D6=0x00000000     D7=0x00000000
```

15.31 RESET

Soft reset the monitor firmware.

USAGE:

reset -[t:x]

DESCRIPTION:

Request that the monitor firmware re-enter at various points in its own context. With no options, the reset is as close to a hard reset as can be supported by the firmware.

OPTIONS:

- -x
  Restart the monitor firmware just as it would be restarted if the application had exited;
- -t {##}
  Restart the monitor with the type specified (primarily for use during monitor development)

15.32 RETURN

Return from subroutine.

USAGE:

return

DESCRIPTION:

Return to the point at which the currently active subroutine was called via gosub..

EXAMPLES:

- Within a script file:

```
# TOPofScript
set CNT 0

# NEXT:
gosub ECHO_CNT
if CNT lt 10 goto NEXT
exit

# ECHO_CNT:
echo $CNT
set -i CNT
sleep 1
return
```

15.33 SET

Set, clear or adjust a shell variable within the monitor.

USAGE:

set -[abcdeF:f:iox] [varname] [value]

DESCRIPTION:

The monitor allows the user to set up shell variables that can then be used by other commands with the monitor or by the client application through the *getenv()* shared library call. This capability provides a clean interface between monitor scripts and applications; thus, allowing the script to define operating characteristics that are activated by the application through the applications use of getenv().

Keep in mind that the shell variables are generic in nature, so they can be used for general-purpose command line substitution as well.

If no arguments are present, *set* will simply dump the current set of environment variables.

OPTIONS:

- -a
  Logically AND the content of the shell variable with the value specified;
- -b ###
  Redefine the console baud rate.
- -c
  Clear the current environment (remove all shell variables);
- -d
  Decrement the content of the shell variable by the value specified (or 1 if value is not specified);
- -e
  Build an environment string; the specified shell variable is loaded with an address in memory that contains a complete list of the current shell variables in a name=value syntax. The syntax of the created string is NAME=VALUE\nNAME=VALUE\n … NAME=VALUE\nNULL.
- -f {filename}
  Build an executable script file that could be used to re-create the current environment;
- -F {filename}
  Interactively build an executable script that will re-create a subset of the current environment;
- -i
  Increment the content of the shell variable by the value specified (or 1 if value is not specified);
- -o
  Logically OR the content of the shell variable with the value specified;
- -x
  Output the result of the -i or -d operation in hex (else decimal).

EXAMPLES:

- set –a ABC 0x00ff
  Logically AND the content of $ABC with 0x00ff
- set –c
  Remove all shell variables
- set –d ABC
  Decrement the content of $ABC by one
- set –d ABC 4
  Decrement the content of $ABC by 4
- set –e ENVP
  Create “Name=VALUE\n” list and load ENVP with the address at which it was placed in memory.

15.34 SLEEP

Delay for a specified number of seconds (or milliseconds).

USAGE:

sleep -[clmv:] {time (seconds/milliseconds) | new loops-per-second count}

DESCRIPTION:

Used within a TFS script, this command simply puts the processor in a delay loop for a specified amount of time. Note that by default, the monitor does not use any CPU facilities for delays and timeouts; hence, it is not truly aware of time because there is no dedicated interrupt handler that has a known frequency. To support the monitor's ability to have a relatively close approximation of a second of delay, the monitor is built with a default loop size (LOOPS_PER_SECOND in config.h). If the sleep time does not appear to be close to the specified number of seconds (or milliseconds), it can be calibrated with the -c option. If no options or arguments are specified, sleep will return the current loops-per-second (LPS) count.

OPTIONS:

- -c
  Calibrate with a new loop value that will be used in subsequent calls;
- -l
  Store the count specified as the new loops-per-second count;
- -m
  Use milliseconds instead of seconds;
- -v {varname}
  Load the specified shell variable with the current LPS count;

EXAMPLES:

- sleep
  Display the current LPS count value.
- sleep 3
  Delay for approximately 3 seconds.
- sleep -m 300
  Delay for approximately 300 milliseconds
- sleep -l 100000
  Redefine the loop count used for the delay to 100000 (non-interactive)
- sleep -c 100000
  Display the old loop count, print out 10 dots each with a delay that corresponds to the new loop count, load 100000 as the new loop count.

NOTES:

- The calibration (-c option) simply outputs 10 dots (.) at a rate specified by the new loopvalue (argument to the -c option). If the rate of the dot output is not 1 per second, adjust the value accordingly. The -l option is simply a non-interactive version of -c. Both -l and -c will modify the loops-per-second count used in subsequent calls. There are a few other internal timeouts that also use the loops-per-second value used by sleep; so adjustment of this duration will adjust other modules' timeout accuracy as well.

15.35 SM

Search memory.

USAGE:

sm -[24cnqsx] {start address} {finish address} {search_value}

DESCRIPTION:

Search through memory for a specified value, or block of data. This value can be a byte, short, long value or a ASCII-coded hex string or straight ASCII string.

OPTIONS:

- -2
  assume the width of the memory accesses is x2
- -4
  assume the width of the memory accesses is x4
- -c
  arg 2 is count
- -n
  search for NOT (not applicable for –s or –x)
- -q
  quit after first successful search hit
- -s
  search_value is a string and the access is byte-wide
- -x
  search-value is a ASCII-coded hex string and the access is byte-wide.

EXAMPLES:

- sm 0x1c000 0x1c100 0x11
  Search through memory range 0x1c000-0x1c100 looking for 0x11
- sm –s 0x1c000 0x1c100 mom
  Search through memory range 0x1c000-0x1c100 looking for the string “mom”

RUNTIME EXAMPLE:

With the following memory at 0x1c000…

```
0001c000: 20 00 0f e0 00 00 00 80   ff 82 6d 94 ff ff ff ff    .........m.....
0001c010: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
0001c020: ff ff ff ff ff ff ff ff   ff ff ff ff ff ff ff ff   ................
0001c030: ff ff ff ff ff ff ff ff   ff ff ff ff 00 00 00 00   ................
0001c040: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................
0001c050: 11 22 33 44 00 00 00 00   00 00 00 00 00 00 00 00   ."3D............
0001c060: 68 65 6c 6c 6f 5f 6d 6f   6d 21 00 00 00 00 00 00   hello_mom!......
0001c070: 00 00 00 00 00 00 00 00   00 00 00 00 00 00 00 00   ................

uMON>sm 0x1c000 0x1c100 0x11
Match @ 0x1c050

uMON>sm -s 0x1c000 0x1c100 mom
Match @ 0x1c066

uMON>sm -qn 0x1c010 0x1c100  0xff
Nomatch @ 0x1c03c (0x0)
```

15.36 STRUCT

Create a structure in memory.

USAGE:

struct [-b:f:v] {statement} [statement1] [statement2] …

DESCRIPTION:

This command allows the user to build a structure in memory using the content of a structure definition file (described below). Applications for this include overlaying (and populating) a structure onto some memory mapped IO or for establishing OS-specific configuration structures prior to turning over control to the OS. Refer to the 'cast' command for the inverse of this (overlay a structure onto memory and display it).

OPTIONS:

- -b {base_address}
  Specify the base address of the structure being created. This will override the content of STRUCTBASE if it is set.
- -f {struct_filename}
  Specify the name of the file to use as the structure reference file. This will override the content of STRUCTFILE if it is set.
- -v
  Additive verbosity used as a mask within the code. As of this writing, it is a 3-bit mask (1-2-4). Bit-1 is user-friendly printouts, bit-2 is for verification of the operation on memory you're trying to perform symbolically, and bit-4 is for debugging the struct command itself. So for a typical user, -v (bit1), -vv (bit2) or –vvv (bits 1 & 2) are the only needed verbosity levels.

STATEMENT SYNTAX:

The command allows the user to insert data into memory based on a formatted structure. Each zero-whitespace-statement, as outlined in the usage string above, is of the syntax:

```
STRUCTURE_ID[=VALUE]
```

where…

STRUCTURE_ID is the structure (or structure member) to be operated on;
VALUE (if specified) is the value (or operation) to be applied to the memory location corresponding to the specified STRUCTURE_ID.

The STRUCTURE_ID is the structure/member specification. The highest or outermost level of the structure is specified by the structure *type* as it is in the structure definition and all inner members are referred to by *name*. So, referring to the following structure definition, if you want to reference the `lval1` member of struct `s1`

```
struct abc {
   long lval1;
   char letters[26];
   long lval2;
}
struct def {
   short sval;
   struct abc s1;
   struct abc s2;
   char c;
}
```

the outermost portion of the STRUCTURE_ID is "def" (i.e. the structure type, not the structure name), then each inner member is specified by the name, not the type. So, to modify the `lval1` member of the `s1` structure above, the STRUCTURE_ID would be "`def.s1.lval1`".

If the "=VALUE" section of the statement is omitted, then the command simply populates shell variables with the offset (STRUCTOFFSET) and size (STRUCTSIZE) of the specified structure id. If "=VALUE" is present, the content of VALUE can be a hard-coded number or one of several different "functions" as follows:

- sizeof(structure_type)
  This returns the size of the specified structure.
- tagsiz(structtype1,structtype2)
  This function is specifically added for support of ARM's ATAGS. Within each tag structure is a a size entry. This size is the sum of the size of two structures divided by 4. So, this is equivalent to the 'C' statement `(sizeof(structtype1) + sizeof(structtype2))/4 .`
- memcpy(address,len)
  This will copy 'len' bytes from 'address' to the offset of the structure member specified by STRUCTURE_ID.
- strcpy(address | "string")
  This will copy the string specified by "string" or the address (requires leading "0x") to the offset of the structure member specified by STRUCTURE_ID.
- strcpy(address | "string")
  Similar to 'strcpy', this will concatenate the string specified by "string" or the address (requires leading "0x") to the offset of the structure member specified by STRUCTURE_ID.
- e2b(MAC_ADDRESS)
  This will convert the ASCII MAC address (hex-colon notation) to a 6-byte binary and transfer it to the offset of the structure member specified by STRUCTURE_ID.
- i2l(IP_ADDRESS)
  Similar to 'e2b', this will convert the ASCII IP address (decimal-dot notation) address to a 4-byte binary and transfer it to the offset of the structure member specified by STRUCTURE_ID.

This command does not support structure tables; however it does support tables of basic types (char, short, long) to a limited degree. When specifying a member that is an array, only the base of that array is accessible; hence referring to the above structure, if I wanted to access the base of the table 'letters', I would specify "`def.abc.letters[26]`" as the STRUCTURE_ID, and that would give me access to the base of that array.

STRUCTURE FILE:
The structure file (specified by –b {fname} or the content of shell var STRUCTFILE) simply contains a structure definition. The syntax is similar to a standard 'C' structure. The first line of the structure declaration must be of the syntax

```
STRUCT_KEYWORD WHITESPACE STRUCTTYPE WHITESPACE OPENBRACE
```

where…

STRUCT_KEYWORD is the string "struct"
WHITESPACE is any combination of spaces (0x20) and tabs (0x08)
STRUCTTYPE is the name of the structure being specified
OPENBRACE is the open curly brace character '{'

This must be on one line. The remaining lines of the definition declare each member of the structure (one per line). The only supported basic types are char, short and long, so those three unit types, plus additional struct declarations are all that is supported within this structure definition. Each line within the structure MUST be of the following syntax:

```
STRUCT_KEYWORD WHITESPACE STRUCTTYPE WHITESPACE STRUCTNAME SEMICOLON
or...
BASIC_KEYWORD WHITESPACE BASICNAME SEMICOLON
```

where…

STRUCT_KEYWORD is the string "struct";
BASIC_KEYWORD is either "char", "short" or "long"

WHITESPACE is any combination of spaces (0x20) and tabs (0x08)
STRUCTTYPE is the name of the structure being specified
STRUCTNAME is the name of the structure;
SEMICOLON is the semicolon character ';'

The final (i.e. closing) line of the structure definition is a closed curly brace as the first (and only) character of the line. Following are a few examples…

```
# 16550-type UART structure:
struct uart {
   char rxdtxd;
   char ien;
   char lctl;
   char lstat;
   char mstat;
   char spr;
}

# DUART structure:
struct uarts {
   struct uart u1;
   struct uart u2;
}
```

The file also allows for empty lines and comments using the '#' as the comment delimiter.
Note that it is possible that the structure definition be placed within the same file (script) as the 'struct' commands. To do this, specify the structure file (-b or STRUCTFILE) as the currently running script file ($ARG0). Then when 'struct' is used, it will access the same script file, but will process ONLY lines that are prefixed by the string "###>>". All of the above sytax is identical, except that now all "structure definition" lines must be prefixed by that string (an example is shown below).

SHELL VARIABLES USED:

- STRUCTFILE:
  This shell variable must be set to point to the name of the file that 'struct' is to use to retrieve the structure definition. If this shell variable is not set, then the –f option must be provided on the command line; otherwise 'struct' will report an error.
- STRUCTBASE:
  This shell variable must be set to the base address of the structure being created by 'struct'. If this shell variable is not set, then the –b option must be specified on the command line; otherwise 'struct' will report an error.

SHELL VARIABLES CREATED:

- STRUCTOFFSET:
  The 'struct' populates this variable with the offset of the specified member, relative to the base structure. If no member is specified, then this will be loaded with zero.
- STRUCTSIZE:
  The 'struct' populates this variable with the size of the specified member. If there is no member specified, then this will be loaded with the size of the base structure.

EXAMPLE1:
Assume the following file called "structs" in TFS…

```
uMON>tfs cat structs
struct hdr {
    long size;
    long tag;
    char mac[6];
    char msg[32];
}
```

The following 'struct' commands will access this file and build the 'hdr' structure at location 0x20000000…

```
uMON>set STRUCTBASE 0x20000000; set STRUCTFILE structs
uMON>struct hdr.size=99
uMON>struct hdr.tag=0x54410001
```

By default there is no output; however, the –v option (bit-1) prints a "friendly" version of what was done…

```
uMON>struct -v hdr.size=99
hdr.size = 99 (0x63)
uMON>struct -v hdr.tag=0x54410001
hdr.tag = 1413545985 (0x54410001)
uMON>
```

The –vv option (bit-2) will show you exactly what's being done in memory…

```
uMON>struct -vv hdr.size=99
*(ulong *)0x20000000 = 45 (0x2d)
uMON>struct -vv hdr.tag=0x54410001
*(ulong *)0x20100004 = 1413545985 (0x54410001)
uMON>
```

There are several 'functions' that can be used on the right-hand side of the equation…

```
uMON>struct hdr.mac=e2b(00:11:22:33:44:55)
```

or better yet…

```
uMON>struct hdr.mac=e2b(${ETHERADD})
```

For establishing strings in a character array, use strcpy and strcat…

```
uMON>struct hdr.msg[32]=strcpy("this is text")
uMON>struct hdr.msg[32]=strcat(" more text appended")
```

EXAMPLE2:
This example shows how the invoking script can contain the structure definition…

```
uMON>tfs cat script
###>>struct hdr {
###>>    long size;
###>>    long tag;
###>>    char mac[6];
###>>    char msg[32];
###>>}
set STRUCTFILE $ARG0
set STRUCTBASE 0x200000
struct hdr.size=44
struct hdr.tag=0x12345
```

If the name of the structure definition file is the currently running script ($ARG0 usually contains the name of the currently running script), then when 'struct' searches through the file it will only parse lines that start with "###>>" (3 consecutive pound signs followed by 2 consecutive right arrows).

NOTES:
- Refer to section 13.2.3 of this manual for more examples and discussion.
- The structure specification on the command line can reference any basic type. If the entry is an array, then the symbol must be the entire array declaration. For example, in the above commands, notice that the

"`line[1024]`" member is shown on the command line as "`tags.cmdline.line[1024]`". The array syntax is a required part of the member name when referenced by the 'struct' command. Similarly, if the entry is a pointer, then the preceding asterisk is a required part of the member name when referenced by the 'struct' command. The struct command does not provide the ability to reference 'within' an array, it can only reference the base address of the array.

- This command does a lot of parsing on both the command line and within the structure definition file. It ***can*** be broken, so don't test it! Use it within the constraints outlined in this text and things should work just fine.
- This command makes no attempt to do any padding. If you build a structure that is a char followed by a long, then that long will be the next location in memory. As a result, you need to be aware of padding as it is needed for the structure you are creating.

15.37 STRACE

Stack trace (used after exception or breakpoint).

USAGE:

strace [-d:F:P:rv]

DESCRIPTION:

This command attempts to use the content of the register cache and the stack pointed to by the registers in that cache to assemble a function nesting. This command is typically used after an exception has occurred and the user needs to determine how the code got to that exception. For best results, this command looks for a symbol table file (see section 10.2.1) to symbolically dump a function name instead of an address within a function.

OPTIONS:

- -d ##
  Specify the maximum depth that strace should nest
- -F ##
  Specify the frame pointer to be used
- -P ##
  Specify the program counter (or instruction pointer) to start with
- -r
  Dump registers
- -v
  Verbose

EXAMPLES:

- strace
  Dumps the current function nesting, showing the function name and the hex offset into the function.
- strace –F 0x123456 –P 0x400010
  Dump a stack frame using 0x123456 as the stack frame pointer and 0x400010 as the program counter. This is useful for running context-sensitive stack traces within a multi-tasking RTOS environment.

NOTES:

- The code behind this command is heavily dependent on the CPU as well as the toolset used to compile the application code. As of this writing (7/14/2004), the *strace* command has been ported to MIPS, ColdFire, ARM, PowerPC & SH2 processors that are compiled with the GNU-based cross-compilation toolsets. Note also that only v2 of the toolsets have been verified to work with these strace commands; however, assuming there has been no change between v2 & v3 regarding stack frame structure, there is no reason why this command will not work with application code built with newer versions of the GNU cross-compilation toolsets.

WARNING:

- The monitor allows small applications to be run directly out of the monitor's own stack space. Be aware that strace will not work for those cases simply because the monitor will be reusing the same stack that the application had been using. Bottom line... if you want to use *strace*, then establish a stack frame for the application.

15.38 SYSLOG

Syslog/UDP message client.

USAGE:

syslog [-f:lnP:p:v] {SRVR_IP} {msg}

DESCRIPTION:

As the name implies, this command is a syslog client. The default port is 514, and the –p & -f options allow the user to specify the syslog priority and facility as outlined by syslog. The basic function is simply to send an ASCII message via UDP to some remote server as some IP/Port address. The –P option allows the user to establish something other than the default 514 port number; hence, this can be used for general purpose UDP messaging with another host on the network.

OPTIONS:

- -f {facility}
  Valid facility values are:

kernel	user	mail	daemon
authorize	syslog	lpr	news
uucp	cron	authpriv	ftp
local0	local1	local2	local3
local4	local5	local6	local7

- -l
  list all valid facility and priority values
- -n
  include the string's null terminator in the UDP message
- -P ##
  Override the default port of 514.
- -p {priority}
  Valid prioity values are:

emerg	alert	critical	error
warning	notice	info	debug

- -v
  Verbose

15.39 TFS

Tiny file system command line interface.

USAGE:

tfs -[d:f:i:mv] {operation} [args]

DESCRIPTION:

TFS provides a small "file-system-like" capability on a target that contains FLASH memory. It provides a core set of commands that allow the user to add, delete, list, display, execute, load, and copy files within flash memory. In addition to the user interface command "TFS" there is also a set of application entry points (or functions) that the application can access through a shared vector table seen by both the monitor and the application. This allows the monitor to store files that can be accessed by the application. It provides a clean way to interface to non-volatile memory and also a means to allow a given application to configure itself based on files in the file system instead of through rebuilds of source code.

File Operations:

- **add** {name} {src_address} {size}
  Create the file named 'name' to contain the data starting at location "src_address of size "size". Options -f and -i can be used to specify the flags and information field associated with the newly created file.
- **base** {filename} {varname}
  Load the shell variable specified by 'varname' with the base address of the file specified by 'filename'.
- **cat** {name}
  Print the specified file. Assumes the file is ASCII.
- **check** [varname]
  Check the sanity of the files stored in TFS by running various tests (like a crc32 on the data and header). If the -d option is specified, then only check files in the specified device. If a variable name is specified, then that shell variable will be loaded with the string "PASS" or "FAIL" based on the result of the FS check.
- **cfg** {start | restore} [{end} [spare_addr]]
  Reconfigure and/or restore the span of TFS within your target's flash space. Prior to uMon1.8, this could only be done once, and wass only applicable to versions of uMon that are built to run directly out of boot flash. As of uMon1.8, assuming the port-specific changes have been made (see porting section), then TFS can be re-configured as needed. Refer to section 12.1 for a thorough example.
- **clean**[r]
  Cleanup (defragment) the file system to free-up flash space. If 'r' is appended then the system will automatically restart after the cleanup. If the -d option is present, then only cleanup the device specified.
- **cp** {name} {newfilename | hex address}
  Copy the named file to the new named file. If the destination begins with '0x', then it is assumed to be a hex address pointing to RAM. The source file is first copied to memory pointed to by APPRAMBASE, then the new file is created from the data in RAM.
- **freemem** [varname]
  Return (or store in 'varname') the amount of flash memory that is still available for use by TFS. If the -d option is specified, then list the memory that is available for that device only. Note that since there is per-file overhead, the value returned here is the amount of data space available if ONE more file is stored in TFS; if additional files are to be stored, then the user must take into account the TFS overhead.
- **info** {filename} {varname}
  Load the shell variable specified by 'varname' with the information field stored with the file specified by 'filename'.
- **init**
  Initialize the file system (remove all files and erase flash).
- **ld** {filename | hex address}
  Load the executable COFF, ELF or AOUT file from flash to RAM. If the file is compressed, ld will automatically decompress. Refer to decompression discussion on TFS. If verbosity is set to something greater than 1 (-vvv), then the load will not touch memory, it will only display what it would load to memory. As of release 1.12 (Mar 2007), the TFS loader can handle loading executable images from any memory address instead of just from a named file. This allows the user to store the image outside of TFS is needbe, but still have the ability to load it as if it was a executable image ('E' flag) in TFS flash space. When a hex

address is used, it must include the file flags as part of the comma delimited text. Note that the “ld” command within TFS will automatically populate the ENTRYPOINT shell variable with the entrypoint address of the image. This allows a script to have a “tfs ld filename” followed by “call $ENTRYPOINT” as an alternate means of launching an executable image.

- **ldv** {filename}
  Verify that the executable binary image in flash space matches what is in RAM space. This only works for files that are not compressed.
- **ln** {src} {linkfilename}
  Link the file specified by “src” to a new linkname specified by “linkfilename”.
- **log** [{on|off} {message}]
  Turn on or off (or determine the current state of) the change-log facility.
- **ls** [filter] [filter...]
  List the current set of files in the file system. There are 4 different levels of verbosity for ls:
  - lvl 0 No verbosity, short list of all active files.
  - lvl 1 Display "hidden" files (beginning with '.') in short format.
  - lvl 2 Display active files in long format.
  - lvl 3 Display active and deleted files in long format.

  Specifying the filter can limit the number of files listed...
  **filter* indicates a suffix match
  *filter** indicates a prefix match
  filter indicates a full filename match
  See notes below for some additional details.

- **qclean** [start] [size]
  Run a ‘quick’ defragmentation on the TFS flash space. This has the advantage of being substantially faster than the standard power-safe defragmentation (hence it is gentler on the spare sector). It has the disadvantage that if the defragmentation is interrupted for any reason (power hit, reset, etc..), then the files in TFS are likely to be corrupted. By default, qclean will copy all valid (non-deleted) files back-to-back in RAM space starting at $APPRAMBASE. To override this default, specify [start] as the base address and [size] as the amount of space allocated at that address.
- **ramdev** {name} {start address} {size}
  Create a temporary ram-based TFS device occupying the ram space designated by {start address} and {size}. The name of the new device (specified by {name}) will automatically be wrapped with two leading slashes and one post slash. For example, if the name specified is TMPRAM, then TFS creates a device named //TMPRAM/.
- **rm** {filter} [filter ...]
  Remove the specified file(s) (see note below). See discussion on "ls" above for details on the filter.
- **rms** {size} [except_file1] [except_file2] ...
  Remove space. Remove files until "size" bytes have been removed. Do not remove any of the "except" files. This is primarily used for testing TFS with scripts.
- **run** {name}
  Run the specified file based on the creation attributes. It will be run as either a script or an executable image. In the case of scripts, the –v option will cause the TFS script runner to print the line of the script prefixed by the line number within the script.
- **size** {filename} {varname}
  Load the shell variable specified by 'varname' with the size of the file 'filename'.
- **stat**
  Dump the current state of TFS. This includes information regarding the location of multiple non-contiguous directories (if any), the amount of space taken up by file overhead, amount of space that would be released if a defragmentation was done, etc...
  For use within a script, this command also generates a set of shell variables for each TFS device it finds. The variables are as follows (where ‘N’ starts with 0 and increments by 1 for each TFS device):

  - TFS_PREFIX_N: Name of TFS device.
  - TFS_START_N: Starting address of TFS device.
  - TFS_END_N: End of TFS device.

- TFS_SPARE_N: Starting address of spare sector.
- TFS_SPARESZ_N: Size of spare sector.
- TFS_SCNT_N: Number of flash sectors in the device.
- TFS_DEVINFO_N: Device info flags.
- TFS_DEVTOT: Total number of TFS devices.

- **trace** [trace-level]
  Establish a runtime trace of the TFS system calls. If no trace level is specified, then the current level is displayed. This tracing is useful for debugging application code that uses TFS system calls...
  - lvl 0 No runtime trace
  - lvl 1 Trace for all TFS system calls except tfswrite(), tfsread() and tfsgetline().
  - lvl 2 Trace for all TFS system calls including tfswrite(), tfsread() and tfsgetline().
  - lvl 3 Trace for all above plus the flash operations.

- **uname** {prefix} {varname}
  Load the shell variable specified by 'varname' with a filename (starting with the specified prefix) that is not currently being used for file storage in TFS.
- **uncmp** {fromname} {to_name | to_addr} [var]
  Uncompress file specified by *fromname* to either another file or address in memory.

OPTIONS:

- -d {device-prefix}
  apply the command to the specified TFS device
- -f {flags}
  flags (see below) applied to the file
- -i {info}
  information field included with the file created
- -m
  enable 'more' when displaying a file or list of files
- -v
  enable verbosity for various tfs operations.

EXAMPLES:
- tfs ls
  Displays the current set of files and their attributes in alphabetical order.
- tfs -v list
  Displays the current set of standard and hidden files (see notes below for more details on the output).
- tfs -vv list
  Displays the current set of files and their data structure verbosely. Note that this list is displayed in the order in which they exist in the file system, not alphabetical.
- tfs -i test -f e add myfile 0xa0100000 459
  Add the file "myfile" to the file system. It will contain a information field of "test" and have attributes "e" (executable script). The actual file will contain 459 bytes of data starting at location 0xa0100000.

NOTES:
- The user must be aware of the fact that the file system is in flash and that there are certain limitations imposed by the underlying technology and microprocessor to memory interface.
- The "tfs ls" output can be formatted with varying levels of additional verbosity (see -v option above). In the simplest case, "tfs ls", the output would look like this:

```
uMON>tfs ls
 Name                      Size   Location   Flags  Info
 //RAM/                           (dir)
 /cfg/                          (dir)
 1linux.cmd                1288  0x102278bc  e
 monrc                       73  0x1034e9bc  e
 ramtst                  209856  0x10381edc  E
```

```
 zImage                    1207092  0x10227e2c

Total: 6 items listed (1418309 bytes).
```

Notice that although there is no directory hierarchy in TFS, filenames can include a slash and the output of this listing will attempt to treat that as if it was a directory. This allows the user to organize filenames in groups. Also, the above case demonstrates the situation where TFS spans across multiple, non-contiguous devices. //RAM/ is a device created by the “ramdev” operation in tfs and all other files are in flash. To see all files regardless of the slash, use "tfs ls *", or "tfs ls cfg/*" for a subset. The size of each file shown is the size of the data space within the flash that is used by the file; this does not include the overhead space needed for each file header and the space needed at the end of TFS flash area for defrag (refer to description of tfs overhead for details on this). The total item count displays the total number of entries listed, some of which may be what the "ls" is displaying as if it was a directory. The total bytes shown at the bottom of this listing also, does not include the overhead or files not displayed because of the slash.

The "tfs -vv" ls output displays more information about each file. This includes the information field, file attributes and other header-related items. The output is not in alphabetical order as it is for the lower level of verbosity and the default filter ignores the slash within the filename. The output in this level is in the order the files are stored in flash. One additional note for this level of verbosity, the total size displayed includes the overhead of the file (header and defrag space needed). Following is an example of the verbose output...

```
uMON>tfs -vv ls
 Name:  '1linux.cmd'
 Info:  ''
 Flags: executable,
 Addr:  0x102278bc (hdr @ 0x10227860, nxtptr = 0x10227dd0)
 Size:  1288 bytes (crc=0xdc572cbf) in sector 17

 Name:  'zImage'
 Info:  ''
 Flags:
 Addr:  0x10227e2c (hdr @ 0x10227dd0, nxtptr = 0x1034e960)
 Size:  1207092 bytes (crc=0xe2d1432a) spanning sectors 17-26

 Name:  'monrc'
 Info:  ''
 Flags: executable,
 Addr:  0x1034e9bc (hdr @ 0x1034e960, nxtptr = 0x1034ea10)
 Size:  73 bytes (crc=0x9e47cc34) in sector 26

 Name:  'ramtst'
 Info:  ''
 Flags: elf_msbin,
 Addr:  0x10381edc (hdr @ 0x10381e80, nxtptr = 0x103b52a0)
 Size:  209856 bytes (crc=0xb940a11) spanning sectors 28-29

 Name:  '/cfg/file1'
 Info:  ''
 Flags:
 Addr:  0x103b52fc (hdr @ 0x103b52a0, nxtptr = 0x103b5310)
 Size:  15 bytes (crc=0x6650af15) in sector 29

 Name:  '/cfg/file2'
 Info:  ''
 Flags:
 Addr:  0x103b536c (hdr @ 0x103b5310, nxtptr = 0x103b5380)
 Size:  17 bytes (crc=0xe750e762) in sector 29

 Name:  '//BBRAM/abc'
 Info:  ''
 Flags:
 Addr:  0x2080005c (hdr @ 0x20800000, nxtptr = 0x20800080)
 Size:  27 bytes (crc=0x81f9fff8)
```

```
Total: 7 accessible files (1419460 bytes).
```

15.40 TFTP

Trivial File Transfer Protocol client/server.

USAGE:

tftp -[aF;f:i:vV] {on|off|IP} {get fname [addr] }

TFTP CLIENT DESCRIPTION:

The tftp command is primarily for use by the client side of the TFTP implementation of MicroMonitor. One exception to this is that the arguments *on* and *off* are used to enable and disable (or shutdown) the tftp server (refer to the server discussion below for more details). This command allows the user to transfer a file from some remote TFTP server into either RAM/DRAM space or into a file in TFS. All transfers initially end up in target ram space. Then, if specified, the -F option informs this client that the data is to be transferred to a file in TFS. By default, the file on the host is initially transferred to the address specified by the $APPRAMBASE shell variable, but the optional addr parameter overrides that default. Upon successful completion of the tftp get command, the shell variable $TFTPGET will be loaded with the amount of data transferred.

Note: This tftp command is primarily for use by the client side of the TFTP implementation of MicroMonitor. One exception to this is that the arguments *on* and *off* are used to enable and disable (or shutdown) the tftp server (refer to the server discussion below for more details).

Packet Retransmission and Timeout Process...

Both the client and server will, under certain circumstances, have to re-transmit a packet that may have been lost. The retry mechanism is based on RFC2131 (DHCP, section 4.1). The initial retry value (retransmit_delay) is doubled until some maximum value (retransmit_delay_max) is reached. At that point the delay is no longer doubled, but some final number of retries (giveup_count) are performed at the rate of that last delay. These three values (retransmit_delay, retransmit_delay_max and giveup_count) have defaults but can be tuned if the shell variable $TFTPRETRYTUNE is set. Note that this timeout/retry mechanism is used by DHCP, TFTP and ARP.

OPTIONS:

- -a
  transfer from host in netascii mode (default is "octet");
- -F {file}
  name of TFS file to copy to
- -f {flags}
  flags assigned to TFS file after copy;
- -i {info}
  file info assigned to TFS file after copy
- -v
  low verbosity, show ticker (per packet transferred)
- -V
  high verbosity, show TFTP opcodes

EXAMPLES:

- tftp -F appfile -f eC -i 11/18/1999 135.3.130.1 get dir1/hostfile
  Retrieve a file from a TFTP server running on a system whose IP address is *135.3.130.1*. The file on the host (relative to the top-level directory of the TFTP server) is called *dir1/hostfile* and it is transferred to the file *appfile* on the target with flags *eC* (executable COFF) and an info field of *11/18/1999*.
- tftp 135.3.130.1 get dir2/another_file
  Retrieve the file dir2/another_file from a host TFTP server at 135.3.130.1 and place it in the location stored in the shell variable $APPRAMBASE.
- tftp 135.3.130.1 get dir3/yanotherfile 0x10000000
  Retrieve the file dir3/yanotherfile from a host TFTP server at 135.3.130.1 and store it at location 0x10000000.

NOTES:

As a client, tftp *put* is not supported (use the server, and a host based client).

TFTP SERVER DESCRIPTION:
First of all...
For the sake of this discussion, realize that we are referring to a situation where the server is the MicroMonitor-based target and the client is some PC or workstation; hence, all of the "tftp" commands mentioned in this section are done on a remote PC or workstation. For transfers from target to host (*tftp get*), the source file is on the target and the destination file is on the host; likewise, for transfers from host to target (*tftp put*), the source file is on the host and the destination file is on the target.
Also...
When the monitor first starts up, if the Ethernet interface is active, it automatically starts up a TFTP server and the monitor will respond to incoming TFTP requests by default. The TFTP server in the monitor looks much like any other TFTP server. Differences are due to the fact that the server allows the client to transfer files several different ways.

From host file to target file:
The client specifies a destination that consists of the destination file name, the flags that are to be associated with the file and the information field that is part of the TFS header for keeping a description of the file. The syntax of this destination is comma delimited:

filename,flags,information_field

All text up to the first comma is considered the filename; text between the first comma and second comma is used as the file flags or attributes, and all text after the second comma is taken as the information field for the TFS file.
Notes:

- the flags field can be empty (2 commas back-to-back) and the information_field need not be specified; or both the flags and information_field can be omitted. Here are a few variations of the above syntax:
  filename,,information_field
  filename,flags
  filename

- when this transfer is actually in progress, the file is first transferred to the location stored in the $APPRAMBASE shell variable; after data transfer completes, it is then copied to TFS flash space.
- if the file already exists in TFS, then it will be deleted and if necessary, a defragmentation will be done.
- if the file is ASCII, then the TFTP server will properly handle "netascii" mode for transferring to the target; the default mode is "octet"

From host file to target RAM:
If the destination file name starts with "0x", then a file on the host can be transferred to the hex address that follows. This memory space is assumed to be standard writeable RAM or DRAM.

From host file to destination assumed to be in a shell variable:
If the destination file name starts with a dollar sign '$' and the string following the '$' is a valid shell variable in the monitor context, then the destination will be taken as whatever is the content of the shell variable. If the shell variable does not exist, then the destination will be that string (a filename that starts with a '$').

From target file to host file:
This is a standard transfer. The file must exist in TFS, and it will be transferred to the host; otherwise, a TFTP error message is sent to the client.

From target memory to host file:
This satisfies the case where a block of memory not necessarily part of TFS needs to transferred up to a host. The source file specification is of the following syntax:

hex_address,length

This will cause the server to transfer 'length' bytes of data starting at 'hex_address' to the host. The server detects this by seeing "0x" as the first two characters in the source filename, and a comma somewhere later in the string.

Command line examples when using a host-based client to talk to the MicroMonitor TFTP server:

- tftp 135.3.94.136 put srcfile tfsfile,eC,Mar_30,1999@11:18
  Send the file "srcfile" to a target at IP address 135.3.94.136. The destination filename on TFS will be "tfsfile" with the flags eC indicating executable COFF, and the information field will contain the string Mar_30,1999@11:18.
- tftp 135.3.94.136 put srcfile 0x10400000
  Send the file "srcfile" to the target at IP address 135.3.94.135. The destination is an address that must be in RAM space of the target.
- tftp 135.3.94.136 put srcfile \$APPRAMBASE
  Send the file "srcfile" to the target at IP address 135.3.94.135. The destination is the content of the shell variable $APPRAMBASE. Note that the '$' is preceded by a backslash. This is because we want the host shell to ignore the '$' and pass it to the target as is. Obviously this, then, depends on the shell running on the host at the time.
- tftp 135.3.94.136 put srcfile
  Send the file "srcfile" to the target at IP address 135.3.94.135. The destination is the same filename in TFS flash space.
- tftp 135.3.94.136 get srcfile
  Retrieve the file "srcfile" from the target at IP address 135.3.94.135. Once on the host it will have the same name.
- tftp 135.3.94.136 get 0x10800400,900 bdata
  Retrieve 900 bytes starting at location 0x10800400 and place them in the file bdata.

15.41 ULVL

User level.

USAGE:

ulvl -[c:hp] [usrlvl | min | max] [password]

DESCRIPTION:

This command is used to set or configure the monitor's user level. The user level of the monitor determines what commands it can execute and what files are accessible. This command is hard coded to require only user-level 0 to execute. All other commands are configurable through the config.h file. For a thorough description of the user level functionality in the monitor refer to the user-level description.

OPTIONS:

- -c {cmd,lvl}
  Modify the user level of command "cmd" to level "lvl". The value of lvl must be between 0 and 3 and if the level is being lowered, then the current user level that the monitor is running at must be at least as high as the command that is being adjusted.
  If the “cmd” string is “ALL”, then all commands are affected.
  If the “lvl” string is “off”, then the command is essentially disabled until the next reset. This allows the user to essentially remove a command from access through the built-in command list; thus, allowing that command to be replaced by a TFS executable command of the same name in TFS.
- -p
  Go into an interactive mode to build the password storage file. This file will contain 3 lines, each of which will be the password for levels 1 through 3. This file is stored with flags "u3" for highest security.
- -h
  Since the backdoor entry into the monitor requires that the user know the MAC address of the system, this option simply dumps the header that contains that information.

EXAMPLES:

- ulvl -c version,2
  Change the user level of the "version" command to 2.
- ulvl -c version,2 -c help,1 -c dm,3
  Change user level of version to 2, help to 1, and dm to 3.

NOTES:

- The user level of the "ulvl" command cannot be adjusted. It must be able to run at user level zero.
- Refer to section 7.14 for an example of how to replace a built-in command with a script by turning off the built-in.

15.42 UNZIP

Decompress (via zlib) a file (or block of memory) to some other block of memory.

USAGE:

unzip –[v:] {source (addr,len | filename)} [destination (addr[,len])]

DESCRIPTION:

If the monitor is built with INCLUDE_UNZIP non-zero (in config.h), then this command comes with the decompression package in the monitor. It provides a CLI-based mechanism for decompressing a file or block of memory to some other block of memory.

OPTIONS:

- -v {varname}
  the shell variable specified by “varname” will be loaded with the size of the decompressed space.

EXAMPLES:

- unzip x.gz
  decompress the file “x.gz” and place it in memory starting at APPRAMBASE.
- unzip 0xfff1240,1248
  decompress a 1248-byte block of memory located at memory address 0xfff1240 and put it in memory starting at APPRAMBASE.
- unzip 0xfff1240,1248 0x300000
- decompress a 1248-byte block of memory located at memory address 0xfff1240 and put it in memory starting at 0x300000.

NOTES:

- This uses the same zlib decompression library as is used by TFS for decompression of the sections of the ld file (ELF, COFF or AOUT).

15.43 VERSION

Display the build date of the monitor (and application) executables.

USAGE:

version [application build info]

DESCRIPTION:

With no arguments, version simply displays the date/time at which the monitor was built. If previously, version was executed with application build information, then that string will be printed also.

15.44 XMODEM

Initiate an XMODEM (or YMODEM) data transfer.

USAGE:

xmodem -[a:BcdF:f:i:ks:t:uvy]

DESCRIPTION:

XMODEM is a very simple protocol that allows files to be transferred between 2 machines that understand the XMODEM protocol. This monitor supports XMODEM because it is small, and universal for probably all terminal emulation packages on the PC. Options allow the user to upload and download, and if enabled, transfer the downloaded file to the file system (see TFS). When downloading to the target, the data is initially placed in system RAM. Upon completion of the download, that downloaded data may be transferred to a file in TFS. By default, the RAM address used is the value established by the monitor at boot time known as APPRAMBASE (stored in the APPRAMBASE shell variable and also displayed on the console at reset).

OPTIONS:

- -a {addr}
  address used to override the default APPRAMBASE download destination address.
- -B
  new boot monitor load (see notes below);
- -c
  use CRC instead of checksum;
- -d
  download a block of data. This, along with other options, allows the downloaded file to be placed in RAM or the file system. Note that even if the downloaded file is destined for the file system, it is initially downloaded to RAM and then copied to the file system, so the address specified must be writeable RAM space;
- -f {flags}
  for downloading to a file in TFS, this specifies the flags that will be assigned to the file after the download has completed
- -F {filename}
  for upload or download, specifies the name of the file to transfer.
- -i {info}
  for downloading to a file in TFS, this specifies the information field that will be assigned to the file after the download has completed;
- -k
  use 1K block size (instead of default 128-byte);
- -s {size}
  since XMODEM transfers in fixed block sizes of 128 bytes, the computed download size of a file is likely to be incorrect. This option allows the user to override the compute size with the value specified by "size".
- -t {address}
  enable tracing to a buffer specified by "address". This is primarily used for debugging Xmodem itself.
- -u
  upload a file or block of data. If the upload is a file then the -F option must also be specified. If upload of raw data, then address and size must be specified on the command line;
- -v
  verify only
- -y
  support the YMODEM extensions to XMODEM.

EXAMPLES:

- xmodem -u -F filename
  Upload the file "filename" from TFS to the host machine.
- xmodem -u -a 0x200400 -s 500
  Upload 500 bytes of data starting at location 0x200400 to the host.

- xmodem -d -a 0x6000
  Download a file from the host to location 0x6000 in memory.
- xmodem -d
  Download a file from the host to the start of application RAM space.
- xmodem -d -F AppFile -f eCB -i 02_24_97 -a 0x6000 -s 23456
  Download a file from host to TFS file 'AppFile'. When the file is created in TFS assign flags 'eCB' (see TFS man page) and information field '02_24_97'. The file will be downloaded into RAM starting at location 0x6000 and the final file size used to place the file in TFS will be 23456 bytes.

NOTES:
- The command requires that either -u or -d be specified (-B implies -d).
- The -v (verify) option allows the user to download a block of data to some address, then invoke the same download with the -v option to verify that the data was transferred correctly.
- The basic XMODEM protocol forces all transfers to be some multiple of 128 bytes. This means the files may have junk at the end of them. For a download to TFS, this can be overridden by supplying the final size with the -s option. Then after the download is completed, instead of using the download size computed by XMODEM, it uses the value specified on the command line.
- The -B option makes it more convenient to rebuild the monitor onboard. **Be careful with this because it will take the binary file transferred and use it to rebuild the boot flash; hence, if the binary file is incorrect, the boot will be corrupted.** When rebuilding the monitor, the sequence of events (without using -B) would be:

  1. uMON> xmodem -d {RAM_address}
  2. Hyperterm: xmodem {monitor-binary}
  3. uMON> flash opw
  4. uMON> flash ewrite {boot_address} {RAM_address} {size_of_monitor-binary}

  This is somewhat error prone, so the -B option automates these steps because it knows the RAM_address, boot_address and the size of the space allocated to the monitor in flash. The above steps are replaced with...

  1. uMON> xmodem -B
  2. Hyperterm: xmodem {monitor-binary}

  After the download completes, xmodem -B will then query the user for approval, at which time a carriage return or *y* approves, and all other characters will abort. Once you give approval, allow time for the bootflash to be reprogrammed. This can take several seconds, depending on the speed of the flash device interface and the size of the monitor binary being programmed.

Chapter 16 MicroMonitor Application Programmer's Interface

MicroMonitor has facilities that are quite useful to the application developer; hence, these facilities are made available through a defined API. This section lists each of the API functions currently available to applications running on top of MicroMonitor. Note that all functions assume the inclusion of the header file "monlib.h", and the source from "monlib.c" , both of which are part of the monitor's common source tree. Depending on the facility used in uMon, other header files such as tfs.h and cli.h may also need to be included with the application that connects to uMon.

The API provided by uMon does not require a linkage to the application. The hookup is done through the use of one well-known-address referred to as the MONCOMPTR (monitor-communication-pointer). Further discussion of this pointer follows; but the important thing to be aware of is that no uMon API call is legal until AFTER the application processes the MONCOMPTR value through monConnect().

Note that if there is a conflict between this text and the API of monlib.h, monlib.h is the overriding authority.

16.1 monConnect()

Connect the application to the monitor.

PROTOTYPE:

void monConnect(int (*moncomptr)(), void (*lock)(), void(*unlock)());

DESCRIPTION:

The monitor and application are two totally separate applications. The application must call this function with the value of the monitor's MONCOMPTR value to establish the necessary hooks so that all of the monitor API functions will be accessible. The value of MONCOMPTR is retrieveable by either the output of "help –i" or the content of the MONCOMPTR shell variable[90]. Refer to text around Listing 22 for more details and a working example.

This function MUST be called prior to attempting access to any other monitor API function.

PARAMETERS:

- int (*moncomptr)();
  This is the dereferenced value contained in the MONCOMPTR shell variable. The MONCOMPTR shell variable contains the value to be used. The actual pointer is passed to monConnect() by reference. For example, if 0x00200040 is the content of the MONCOMPTR shell variable, then the following line would be used as the call to monConnect. Notice the asterisk prior to the moncomptr value used to dereference the value…

```
monConnect((int(*)())(*(unsigned long *)0x00200040),(void *)0,(void *)0);
                      ^
          asterisk-----|
```

- void (*lock)();
  Lock function to provide reentrancy protection in a multitasking environment. This can be NULL if not needed.
- void (*unlock)();
  Unlock function for reentrancy protection. This can be NULL if not needed.

NOTES:

- For a detailed discussion on the use of the *lock* and *unlock* functions refer to section 8.8.
- It is possible that the version of the monitor on the target is older than the version of the monitor linkage files (monlib.c & monlib.h) that are being used with the application to connect it to the monitor. This call to monConnect() may generate some warning messages that inform the user that the on-board monitor does not support some set of API hooks that are newer than the on-board monitor. As long as the identified hooks are not used in the application, there is no problem; however, it is still best to update the monitor so that these warnings are eliminated. The warning message is as follows (where 0xNNNN is one of the values in monlib.h):

```
moncom unknown command: 0xNNNN
```

RETURN:

void

[90] Prior to uMon1.0, this value was retrieved through the "mstat" command. The "mstat" command has been removed from uMon1.0; however the output of mstat is still available via "help –i".

16.2 mon_addcommand()

Add an application-specific command list to the monitor.

PROTOTYPE:

int mon_addcommand(struct monCommand *cmd, char *usrlvltbl);

DESCRIPTION:

This function allows the user to append a second command table to the end of the monitor's default command table. This is useful for several reasons:

1) If the application is using the monitor's CLI for it's user interface, this provides a very convenient mechanism for adding application-specific commands without eliminating the ability to access the monitor's own command table.
2) If the system crashes, and control is returned to the monitor, this function (or a wrapper) can be called to re-install the commands into the monitor's table so that they can be used to help debug the reason for the crash.

.

PARAMETERS:

- struct monCommand *cmd;
  Pointer to a table of command structures to be added to the monitor's internal command table. The monitor assumes that the final element in the table has a NULL name member.
- char *usrlvltbl;
  Pointer to a table whose user-level entries will correspond to the command table for establishment of the user level for each new command in the table.

RETURN:

0 if successful; else -1.

CODE SNIPPET (refer to section 9.4 for a complete working example):

```
#include "monlib.h"

char *mycmdHelp[] = {                    // Help text array for mycmdFunc()
       "really doesn't do anything",     // Command description
       "[echo string]",                  // Command usage syntax
       0,
};

int
mycmdFunc(int argc,char *argv[])
{
       return(CMD_SUCCESS);
}

struct monCommand mycmdTbl[] = {         // App-specific command table.
  { "mycmd",  mycmdFunc,    mycmdHelp, 0 },
  { 0,0,0,0 }
};

char mycmdUlvltbl[] = { 0 };             // App-specific user-level table.

int
main(int argc,char *argv[])
{
       mon_addcommand(mycmdTbl,mycmdUlvltbl);
       ...
```

16.3 mon_appexit()

Allows the application to return control to the monitor.

PROTOTYPE:

void mon_appexit(int exitval);

DESCRIPTION:

This API call allows an application that to return control to MicroMonitor. Note this is not a simple return to the execution thread that uMon was running prior to turning over control to the application; it is more like a soft restart of uMon, to terminate the application and give control back to MicroMonitor. The code behind this function is somewhat target specific; however, in general this function does what needs to be done to assure that MicroMonitor can interface with the user. It re-initializes the IO that MicroMonitor initializes at startup; however it does not clear BSS space. As a result, any resources that may have been allocated by the application (files, heap space, environment variables, etc…) remain intact.

This API is typically only needed if the application has established its own stack space. If the application simply runs on the stack space of uMon, then the mon_appexit() can be replaced with a simple return.

Refer to sections 9.3.1 and 9.6 above for working examples.

PARAMETERS:

- int exitval
  The value considered to be the exit status. This value is printed by the monitor when this function is called.

RETURN:

void (This function does not return to the caller application)

16.4 mon_com()

This function is the lowest-level hook provided by the monitor so that the application can connect. In general, it is used by the monConnect() function; however, there are a few cases where application code will also use this. Refer to section 8.5 for a working example of these cases.

PROTOTYPE:

int mon_com(int cmd, void *arg1, void *arg2, void *arg3);

DESCRIPTION:

This function provides the basic interface needed for hookup between application and monitor.

PARAMETERS:

- int cmd
  Operation to be performed. The complete list is found in the monitor source file monlib.h.
- arg{1-3}
  Operation-specific arguments.

RETURN:

0 if successful; else –1.

16.5 mon_cprintf()

A centered version of mon_printf().

PROTOTYPE:

int mon_cprintf(char *format, ...);

DESCRIPTION:

Provides the application with a small and simple printf() with limited formatting capability. The text in the format is automatically centered across an 80-character line. Refer to mon_sprintf() (section 16.45) for formatting characters supported.

PARAMETERS:

- char *format
  Pointer to a format buffer
- argN
  Variable argument count list referred to (if any) by format buffer.

RETURN:

The size of the final string printed out the console port.

CODE SNIPPET:

The following line…

```
mon_cprintf("hi mom!");
```

generates the following output…

```
                                    hi mom!
```

(where “hi mom!” is centered across an 80-character screen width)

16.6 mon_xcrc16()

CRC16 calculation.

PROTOTYPE:

unsigned short mon_crc16(char *buffer, long nbytes);

DESCRIPTION:

Perform a crc16 calculation across the specified buffer, using the same CRC16 polynomial that is used with Xmodem.

PARAMETERS:

- char *buffer
  Data space over which the crc16 calculation is to be performed.
- long nbytes
  Size of the buffer..

RETURN:

The calculated crc.

16.7 mon_crc32()

CRC32 calculation.

PROTOTYPE:

unsigned long mon_crc16(char *buffer, long nbytes);

DESCRIPTION:

Perform a crc32 calculation across the specified buffer.

PARAMETERS:

- char *buffer
  Data space over which the crc32 calculation is to be performed.
- long nbytes
  Size of the buffer..

RETURN:

The calculated crc.

16.8 mon_decompress()

Decompress a block of data that has been previously compressed.

PROTOTYPE:

int mon_decompress(char *src, int srcsize, char *dest);

DESCRIPTION:

The monitor incorporates the zlib decompression libraries. This function allows the application to decompress memory (or a data file) that was previously compressed with zlib-based tools (e.g. gzip) and installed on the target.

PARAMETERS:

- char *src
  Pointer to starting location of compressed data.
- int srcsize
  Size of the block of compressed data in bytes.
- char *dest
  Point to location into which the decompressed data is to be placed.

RETURN:

The size of the decompressed data or -1 if failure.

16.9 mon_delay()

A millisecond-resolution delay loop..

PROTOTYPE:

void mon_delay(int msecs);

DESCRIPTION:

This function provides a delay loop. **Be aware that it may not be not precise.** As of uMon1.0, the monitor can be built with a hardware-based timer or a simple (estimated) loop count based timer. The user can determine how the monitor was built through the "sleep" command. If it is configured with a simple (inaccurate) loop count, then the "-c" option will be available for the user to adjust the loop counter. If the hardware-based timer is used, then this option will not be available because it is assumed that the hardware-derived source is accurate. Refer to section 15.34 for more informatino on the sleep command.

PARAMETERS:

- int msecs
  Number of milliseconds to delay.

RETURN:

void

16.10 mon_docommand()

A mechanism by which the application can invoke a command that is part of the monitor.

PROTOTYPE:

int mon_docommand(char *cmd_string, int verbose);

DESCRIPTION:

Similar to the "*system*" function in DOS and/or Unix, this command allows the application code to execute commands that are normally executable from the command line interface of the monitor. This allows, for example, an application to have a console interface with its own set of commands; then if the command entered does not match any of those commands in the application's command table, the command string can be passed to the monitor to see if the command is actually a monitor command. The end result is that the application code can inherit the command table of the monitor with almost zero overhead. Refer to section 10.1 above for a working example.

PARAMETERS:

- char *cmd_string;
  String of characters that would be typed as if they were a command entered at the console interface.
- int verbosity
  If non-zero, the monitor's command interpreter will print the command string after processing the line for environment substitutions (shell variables & symbols).

RETURN:

These return values are found in cli.h.

- CMD_SUCCESS:
  Everything worked ok.
- CMD_FAILURE:
  Command parameters were valid, but command itself failed for some other reason.
- CMD_LINE_ERROR:
  Command line itself was invalid. Too many args, invalid shell var syntax, etc.. Some kind of command line error prior to checking for the command name-to-function match.
- CMD_ULVL_DENIED:
  Command's user level is higher than current user level, so access is denied.
- CMD_PARAM_ERROR:
  Command line did not parse properly. There was a syntax error on the command line that did not even allow the command function to get going.
- CMD_NOT_FOUND:
  Since these same return values are used for each command function plus the docommand() function, this error indicates that docommand() could not even find the command in the command table.

16.11 mon_flasherase()

Erase a sector of flash using MicroMonitor's flash operations.

PROTOTYPE:

void mon_flasherase(int sector_number);

DESCRIPTION:

This function allows the application to use the flash access functions already built into the monitor. If TFS is enabled in the system, then the user must be very cautious regarding flash. Typically, if these functions are used, then certain flash sectors have been omitted from TFS space and are accessible exclusively by these access functions.

PARAMETERS:

- int sector_number
  Sector number of the flash that is to be erased. The output of "flash info" dumps the address space and sector number information.

RETURN:

Negative if failure; 0 if sector is locked or protected; else 1 for success.

16.12 mon_flashinfo()

Retrieve information (base address and size) of a specified flash sector.

PROTOTYPE:

int mon_flashinfo(int sector_number, int *size, char **base);

DESCRIPTION:

This function allows the application to use the flash access functions already built into the monitor. If TFS is enabled in the system, then the user must be very cautious regarding flash. Typically, if these functions are used, then certain flash sectors have been omitted from TFS space and are accessible exclusively by these access functions

PARAMETERS:

- int sector_number
  The sector from which the information is to be retrieved.
- int *size
  If successful and size parameter is non-zero, the size of the sector is loaded into the integer pointed to by this parameter.
- char **base
  If successful and base parameter is non-zero, the base address of the sector is loaded into the char * pointed to by this parameter.

RETURN:

-1 if failure; else 0.

16.13 mon_flashwrite()

Write a block of data to a block of flash.

PROTOTYPE:

int mon_flashwrite(char *destination, char *source, int bytecnt);

DESCRIPTION:

This function allows the application to use the flash access functions already built into the monitor. If TFS is enabled in the system, then the user must be very cautious regarding flash. Typically, if these functions are used, then certain flash sectors have been omitted from TFS space and are accessible exclusively by these access functions

PARAMETERS:

- char *destination
  The location in the flash into which the block is to be written.
- char *source
  The location from which the flash is to be written.
- int size
  The size of the block of data to be written.

RETURN:

-1 if failure; else 0.

16.14 mon_free()

Similar to standard free().

PROTOTYPE:

void mon_free(char *buf);

DESCRIPTION:

Used to release memory that was previously allocated by mon_malloc() or mon_realloc(). If the pointer passed to mon_free() is not a pointer that was previously returned by mon_malloc() an error will result. Refer to mon_malloc() (section 16.29) for other details.

PARAMETERS:

- char * buf
  Pointer to a buffer that was previously allocated by mon_malloc() or mon_realloc().

RETURN:

void

16.15 mon_getargv()

Retrieve an argument list.

PROTOTYPE:

void mon_getargv(int *argc, char ***argv);

DESCRIPTION:

This function provides the hook needed by the application to retrieve an argument list that was created previously by command line invocation. The integer pointed to by argc is loaded with the argument count and the pointer to a character array is loaded with the location of the char *argv[] table. Refer to section 9.2.1 above for a working example.

PARAMETERS:

- int *argc;
  Pointer to an integer that is to be loaded with the number of arguments.
- char ***argv
  Pointer to an array of string pointers that is loaded with the char *argv[] table.

RETURN:

void

16.16 mon_getbytes()

Retrieve some number of bytes from the console port.

PROTOTYPE:

int mon_getbytes(char *buf, int count, int block);

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still read in characters from the serial port designated as the console.

PARAMETERS:

- char *buf
  Pointer to a buffer into which mon_getbytes() will place the characters
- int count
  Number of characters to retrieve.
- int block
  If set, then wait for 'count' characters; else return as soon as there are no more characters present.

RETURN:

The number of characters retrieved.

16.17 mon_getchar()

Provide similar functionality as standard getchar().

PROTOTYPE:

int mon_getchar(void);

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still receive characters from the console port. This function will block waiting for the next incoming character into the console port. If one is already there prior to the call, then the return is immediate.

RETURN:

The character read on the console port.

16.18 mon_getenv()

Similar to standard getenv().

PROTOTYPE:

```
char    *mon_getenv(char *varname);
```

DESCRIPTION:

The monitor can establish shell variables at the command line using the *set* command or by another application using the mon_setenv() API. This function allows an application to retrieve the value that corresponds to a specified shell variable name.

PARAMETERS:

- char *varname
  The name of the shell variable of which to retrieve the content.

RETURN:

Pointer to a NULL-terminated string representing the content of the shell variable specified by varname, or (char *)NULL if the variable does not exist.

16.19 mon_getenvp()

Retrieve a pointer to a string that contains one white space (newline) delimited "name=value" pair for each shell variable currently defined in the system.

PROTOTYPE:

char *mon_getenvp(void);

DESCRIPTION:

The monitor can establish shell variables at the command line using the *set* command or by another application using the mon_setenv() API function. To retrieve the content of just one known environment variable, use mon_getenv(char *varname) (section 16.18). To retrieve a string that contains all “name=value” pairs, use this function. The format of the returned string is:

```
NAME=VALUE NL NAME=VALUE NL NAME=VALUE NL NAME=VALUE NL NULL
```

where

- NL is the newline character
- NULL is the terminating character
- The number of "name=value" pairs is limited only by the amount of heap space available in the monitor. Note that this command does a mon_malloc() to allocate space for the string. It is the responsibility of the caller to free that space (mon_free()) when finished with the string.

RETURN:

The string as specified above or (char *)NULL if no shell variables are set.

16.20 mon_getline()

Retrieve characters from console.

PROTOTYPE:

int mon_getline(char *buffer, int max, int ledit);

DESCRIPTION:

Retrieve characters from the console port until a carriage return or line feed is received (or 'max' characters are received). The ‘ledit’ option enables the use of the monitor’s command line editing facilities.

PARAMETERS:

- char *buffer
  A pointer to the space into which the incoming characters are to be placed.
- int max
  The maximum number of bytes to place into the buffer.
- int ledit
  If set to 1, then enable the use of the monitor's line-editing (vt100 or ksh-vi, depending on config.h) functions while retrieving the line.

RETURN:

The number of bytes retrieved.

16.21 mon_getsym()

Retrieve a symbol from the monitor's symbol table file.

PROTOTYPE:

char *mon_getsym(char *symname, char *buf, int bufsize);

DESCRIPTION:

The monitor may have symbols. The presence of symbols is determined by the presence of as symbol table file usually called "symtbl". This function allows an application to retrieve the value that corresponds to a specified symbol name..

PARAMETERS:

- char *symname
  The name of the symbol whose content is to be retrieved
- char *buf
  Block of memory into which the line of the symbol file will be placed
- int bufsize;
  Size of the line buffer

RETURN:

The content of the symbol specified by symname, or (char *)NULL if the symbol (or symbol file) does not exist .

CODE SNIPPET:

```
int
main(int argc, char *argv[])
{
    if (argc == 2) {
        char buf[64], *cp;

        if ((cp = mon_getsym(argv[1],buf, sizeof(buf))) != 0)
            mon_printf("sym '%s' = '%s'\n", argv[1],cp);
        else
            mon_printf("sym '%s' = ???\n", argv[1]);
    }
    return(0);
}
```

16.22 mon_gotachar()

Return status of character presence on console port.

PROTOTYPE:

int mon_gotachar(void);

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still query for the presence of a character on the console. This query does not affect the status of the console's incoming character stream.

RETURN:

1 if there is a character present; else 0.

16.23 mon_heapextend()

Extend the basic heap that is statically allocated to the monitor..

PROTOTYPE:

int mon_heapextend(char *base, int size);

DESCRIPTION:

The monitor has a heap for its own use of malloc. The memory allocation used by the monitor supports a non-contiguous heap. This means that a second chunk of memory can be given to the monitor for additional heap space and it does not have to be memory contiguously appended to the end of its first allocated block. This API call allows an application to allocate a second chunk of memory to the monitor's heap. For details on the monitor's heap expansion facility, refer to the section on extending the monitor's heap (section 8.7).

PARAMETERS:

- char *src
  Pointer to the starting location of a new block of heap memory space.
- int size
  Size of the block of memory. If this size is set to -1, then the current heap expansion is released.

RETURN:

0 if the request was accepted; else -1 indicating a failure.

16.24 mon_i2cctrl()

I-Squared-C control function.

PROTOTYPE:

int mon_i2cctrl(int interface, int command, unsigned long arg1, unsigned long arg2);

DESCRIPTION:

If the target has an I-Squared-C interface controller and it is hooked into uMon, then this API supports control functions to the interface(s). This is not a standard API for all targets; however, it is generic enough in nature that it is supported under the MicroMonitor API. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port.***

PARAMETERS:

- int interface
  Since there may be more than one I-Squared-C controller on the target, this parameter provides the API with the ability to control each interface uniquely. If there is only one, then this value is simply 0.
- int command
  This is interface specific, but currently the only command supported is I2CCTRL_INIT.
- unsigned long arg1
  The first command-specific argument.
- unsigned long arg2
  The second command-specific argument.

RETURN:

Generally, the function will return negative if the operation fails and zero for success; however, the return value is command-dependent.

COMMANDS:

I2CCTRL_INIT:

arg1 & arg2 are not used. This simply initializes the interface. Typically this initialization will have already been done by the monitor at bootup.

16.25 mon_i2cread()

I-Squared-C read function.

PROTOTYPE:

int mon_i2cread(int interface, int dev_addr, uchar *data, int data_len);

DESCRIPTION:

If the target has an I-Squared-C interface controller and it is hooked into uMon, then this API supports control functions to the interface(s). This is not a standard API for all targets; however, it is generic enough in nature that it is supported under the MicroMonitor API. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port.***

PARAMETERS:

- int interface
  Since there may be more than one I-Squared-C controller on the target, this parameter provides the API with the ability to write to each interface uniquely. If there is only one, then this value is simply 0.
- int dev_addr
  Specify the address of the device to be read from. The upper bits of this field are also used to specify a few different flags that invoke different read options. The flag that may be used by mon_i2cread is REPEATED_START. If this bit is set, then the i2cread function will assume that the "data" pointer has some write data. The first byte of the data is the length followed by the number of applicable bytes. For an example of this, refer to the “i2c” command source code in the monitor.
- unsigned char *data
  Pointer to the block of data to be read from the device.
- int data_len
  Size of the block of data to be read from the device.

RETURN:

This function will return negative if the operation fails else it returns the number of bytes read.. Typically the value returned is equal to the data_len parameter if all goes well.

16.26 mon_i2cwrite()

I-Squared-C write function.

PROTOTYPE:

int mon_i2cwrite(int interface, int dev_addr, uchar *data, int data_len);

DESCRIPTION:

If the target has an I-Squared-C interface controller and it is hooked into uMon, then this API supports control functions to the interface(s). This is not a standard API for all targets; however, it is generic enough in nature that it is supported under the MicroMonitor API. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port.***

PARAMETERS:

- int interface
  Since there may be more than one I-Squared-C controller on the target, this parameter provides the API with the ability to write to each interface uniquely. If there is only one, then this value is simply 0.
- int dev_addr
  Specify the address of the device to be written to.
- unsigned char *data
  Pointer to the block of data to be written to the device.
- int data_len
  Size of the block of data to be written to the device.

RETURN:

This function will return negative if the operation fails else it returns the number of bytes written. Typically the value returned is equal to the data_len parameter if all goes well.

16.27 mon_intsoff()

Allows the application to turn off interrupts.

PROTOTYPE:

unsigned long mon_intsoff(void);

DESCRIPTION:

Allows the application to turn off interrupts. Note that this is probably not useful if you are running with some vendor-supplied OS. Use the function that is supplied with the package. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port.***

RETURN:

The value returned can be used as an argument to intsrestore() to re-establish previous interrupt state.

16.28 mon_intsrestore()

Allows the application to restore interrupt state.

PROTOTYPE:

void mon_intsrestore(ulong ival);

DESCRIPTION:

Allows the application to restore interrupts. Note that this is probably not useful if you are running with some vendor-supplied OS. Use the function that is supplied with the package. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port.***

PARAMETERS:

- unsigned long ival
  CPU-specific value that was previously returned by intsoff().

16.29 mon_malloc()

Similar to standard malloc().

PROTOTYPE:

```
char    *mon_malloc(int size);
```

DESCRIPTION:

This provides the application with a memory allocator whose heap is maintained by the monitor. In addition, the heap is queriable through the monitor's "heap" command (section 15.18). At each allocation/deallocation, the entire control structure of the heap is checked for sanity. Also, in addition to control verification, overrun and under-run checks are made because the space returned by mon_malloc also has additional wrapper space that is checked to verify that the space is not used beyond (or prior to) its limit. This implies overhead which may or may not be desired, but does provide a fairly strong amount of debugging.

The monitor is built with an amount of heap space that is needed for the monitor itself plus a bit extra for application space. The actual amount of extra heap depends on the port. To allow additional memory allocations to be made, the heap command (or the mon_heapextend() API function) allows the user to define a starting point and size of the additional space. Refer to the discussion on extending the monitor's heap (section 8.7) for more information on this.

PARAMETERS:

- int size
  The size of the block of memory to be allocated.

RETURN:

A pointer to the block of memory or (char *)NULL if error.

16.30 mon_memtrace()

Similar to the mon_printf() API but in coordination with the monitor's "*mtrace*" command, the formatted string is placed into a circular buffer in RAM. Refer to section 10.4 above for details on the use of the *mtrace* command and mon_memtrace().

.

PROTOTYPE:

void mon_memtrace(char *format, ...);

DESCRIPTION:

This function allows the application to redirect "printf-like" formatted strings to a RAM buffer area that can be dumped later (see mtrace command, section 15.26). The purpose is to allow the application to use formatted output, but not be concerned with the status of the console device (UART). The strings are formatted and placed in a memory buffer that was previously established by the "*mtrace cfg*" command. The formatting supported is the same as the mon_sprintf() function, so refer to that for limitations.

.

PARAMETERS:

- char *format;
  Pointer to a format buffer (same as printf)
- argN;
  Variable number of arguments.

RETURN:

Number of bytes printed.

16.31 mon_pcicfgread()

PCI config-space read function.

PROTOTYPE:

unsigned long mon_pcicfgread(int interface, int bus, int device, int func, int regno);

DESCRIPTION:

If the target has a PCI interface controller and it is hooked into uMon, then this API supports reading from configuration space in devices on the interface(s). This is not a standard API for all targets; however, it is generic enough in nature that it is supported under the MicroMonitor API. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port***.

PARAMETERS:

- int interface
  Since there may be more than one PCI controller on the target, this parameter provides the API with the ability to control each interface uniquely. If there is only one, then this value is simply 0.
- int bus
  The bus number (usually zero) of the device on the PCI interface.
- int device
  The device number to be accessed on the PCI bus.
- int func
  The function number on the device.
- int regno
  The register number of the configuration space to be modified. This corresponds to an offset into the configuration space. For example, regno0 is offset 0, regno1 is offset 4, regno2 is offset 8, etc..

RETURN:

The function will return the content of the register (offset into config space) that was read.

16.32 mon_pcicfgwrite()

PCI config-space write function.

PROTOTYPE:

int mon_pcicfgwrite(int interface, int bus, int device, int func, int regno, ulong value);

DESCRIPTION:

If the target has a PCI interface controller and it is hooked into uMon, then this API supports writing to configuration space in devices on the interface(s). This is not a standard API for all targets; however, it is generic enough in nature that it is supported under the MicroMonitor API. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port***.

PARAMETERS:

- int interface
  Since there may be more than one PCI controller on the target, this parameter provides the API with the ability to control each interface uniquely. If there is only one, then this value is simply 0.
- int bus
  The bus number (usually zero) of the device on the PCI interface.
- int device
  The device number to be accessed on the PCI bus.
- int func
  The function number on the device.
- int regno
  The register number of the configuration space to be modified. This corresponds to an offset into the configuration space. For example, regno0 is offset 0, regno1 is offset 4, regno2 is offset 8, etc...
- unsigned long value
  This is the 32-bit value to be written to the specified configuration space offset.

RETURN:

The function will return negative if the operation fails and zero for success.

16.33 mon_pcictrl()

PCI control function.

PROTOTYPE:

int mon_pcictrl(int interface, int command, unsigned long arg1, unsigned long arg2);

DESCRIPTION:

If the target has a PCI interface controller and it is hooked into uMon, then this API supports control functions to the interface(s). This is not a standard API for all targets; however, it is generic enough in nature that it is supported under the MicroMonitor API. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port.***

PARAMETERS:

- int interface
  Since there may be more than one PCI controller on the target, this parameter provides the API with the ability to control each interface uniquely. If there is only one, then this value is simply 0.
- int command
  This is interface specific, but currently the only command supported is PCICTRL_INIT.
- unsigned long arg1
  The first command-specific argument.
- unsigned long arg2
  The second command-specific argument.

RETURN:

Generally, the function will return negative if the operation fails and zero for success; however, the return value is command-dependent.

COMMANDS:

PCICTRL_INIT:

arg1 & arg2 are not used. This simply initializes the interface. Typically this initialization will have already been done by the monitor at bootup.

16.34 mon_portcmd()

An API call that is 100% port-specific, defined by the port originator/maintainer.

PROTOTYPE:

int mon_portcmd(int cmd, void *arg)

DESCRIPTION:

This function provides the uMon API with an API call that is 100% port-specific. If included in a build, then the INCLUDE_PORTCMD define must be set to 1. Can't say too much about it. It's just a generic interface that allows the user to define port-specific functionality that is callable through the uMon API.

PARAMETERS:

- int cmd
  Used to define the command type (if more than one) for the port-specific feature.
- void *arg
  Argument, again, specific to the purpose of the command.

RETURN:

The purpose of the return value depends on the purpose of the call..

16.35 mon_printf()

Similar to printf() but limited in the formatting capability.

PROTOTYPE:

int mon_printf(char *format, ...);

DESCRIPTION:

Provides the application with a small and simple printf() with limited formatting capability. Refer to mon_sprintf() (section 16.45) for formatting characters supported.

PARAMETERS:

- char *format
  Pointer to a format buffer
- argN
  Arguments referred to (if any) by the format buffer).

RETURN:

The size of the final string printed out the console port.

16.36 mon_printmem()

Print a block of memory.

PROTOTYPE:

int mon_printmem(char *base_address, int size, int ascii);

DESCRIPTION:

Provides the application with a simple interface to display a block of memory to the console port in ASCII-coded hex (similar to the output of the 'dm' command). Refer to the 'dm' command for an example of the output.

PARAMETERS:

- char *base_address
  Starting point of the memory block to be displayed.
- int size
  Size of the block of memory to be printed.
- int ascii
  If set to 1, then the memory is displayed in ASCII as well as ASCII-code hex. For each character that is non- printable, the ASCII output will display it as a dot.

RETURN:

The size of the block printed.

16.37 mon_printpkt()

Provides the application with the ability to verbosely print out an ethernet packet by using the same code that is in uMon when ether –V (ethernet with verbosity) is enabled.

PROTOTYPE:

void mon_printpkt(char *buffer, int size, int incoming);

DESCRIPTION:

This is useful when debugging an application-based ethernet driver. It simply taps into uMon's ability to dump ethernet packets with some level of intelligence.

PARAMETERS:

- char *buffer
  Pointer to the ethernet packet.
- int size
  Size of the ethernet packet (in bytes).
- int incoming
  Set to 1 if packet is incoming, else 0.

mon_profiler()

Call the monitor's profiling facility.

PROTOTYPE:

#include “monprof.h”

void mon_profiler(struct monprof *mpp);

DESCRIPTION:

The run-time profiling capability of the monitor is used by calling mon_profiler() through some high-priority periodic interrupt handler. The function is called with a pointer to a structure (monprof) that contains either (or both) the instruction address at time of the interrupt and/or the current task ID at the time of the interrupt.

.

PARAMETERS:

- struct monprof *mpp
  Refer to section 10.8 above for details.

RETURN:

void.

16.38 mon_putchar()

Provide similar functionality as standard putchar().

PROTOTYPE:

int mon_putchar(char c);

DESCRIPTION:

Allows the application to be unaware of the console port interface, but still transfer characters to the console.

PARAMETERS:

- char c

  The character destined for the console port of the target.

RETURN:

The same character that was passed to the function is returned.

16.39 mon_realloc()

Reallocate a block of memory from the monitor's heap.

PROTOTYPE:

char *mon_realloc(char *buf, int newsize);

DESCRIPTION:

Allows the application to adjust the size of a block of memory that was previously allocated from the monitor's heap (typical warnings apply with regard to using this in an embedded environment). The new block of memory may or may not be moved relative to the original block. Refer to mon_malloc() (section 16.29) for more details.

PARAMETERS:

- char * buf;
  Pointer to the current buffer;
- int newsize;
  Size to make the new buffer.

RETURN:

If successful, a non-zero pointer will be returned; else NULL..

16.40 mon_recvenetpkt()

Receive a block of data (i.e. ethernet packet) from the connected network interface.

PROTOTYPE:

int mon_recvenetpkt(char *packet, int size);

DESCRIPTION:

This function provides a standalone uMon application with the ability to check for (and receive if available) data from the same network interface that uMon uses prior to turning control over to the application. The incoming packet is ethernet plus payload that is application specific.

It assumes that the underlying version of uMon has an ethernet interface, and simply uses the port-specific polletherdev() function from that port. As a result, this function is likely to only work in a polled environment (since that's what uMon works with). An example of an application that uses this functionality to communicate via ARP/ICMP/UDP with another target can be found in the umon_apps/udp directory which comes with the MicroMonitor tarball.

Refer to mon_sendenetpkt() for additional information.

PARAMETERS:

- char *packet
  Pointer to the buffer area into which the incoming packet will be placed.
- int size
  Maximum size of the incoming packet.

16.41 mon_restart()

Allows the application to re-start the monitor.

PROTOTYPE:

void mon_restart(int val);

DESCRIPTION:

Allows the application to re-enter the monitor at various points. ***The user must be aware that this API call is VERY target-specific; hence, it is heavily dependant on the implementation of the port.***

PARAMETERS:

- int val
  Restart value

RETURN:

void This function does not return.

16.42 mon_sendenetpkt()

Transfer a block of data (i.e. ethernet packet) over the connected network interface.

PROTOTYPE:

int mon_sendenetpkt(char *packet, int size);

DESCRIPTION:

This function provides a standalone uMon application with the ability to transfer data over the same network interface that uMon uses prior to turning control over to the application. The packet is assumed to be a complete ethernet packet with a payload that is application specific.

It assumes that the underlying version of uMon has an ethernet interface, and simply uses the port-specific sendBuffer function from that port. As a result, this function is likely to only work in a polled environment (since that's what uMon works with). An example of an application that uses this functionality to communicate via ARP/ICMP/UDP with another target can be found in the umon_apps/udp directory which comes with the MicroMonitor tarball.

Since this is just a "reuse" of the underlying packet interface, it should work on all ports regardless of the interface (that's the intent). As a result of this reuse, the application must initially inform uMon that it is using the interface. Then, when (if) the application terminates, it should once again inform uMon that it is done. This is done with two special cases of this function:

mon_sendenetpkt(0,0): this tells uMon that the application is going to use the interface; hence, uMon code will not poll the interface internally.

mon_sendenetpkt(0,-1): this tells uMon that the application is done using the interface; hence, uMon returns to its normal polling of the interface.

PARAMETERS:

- char *packet
  Pointer to the ethernet packet starting with the destination & source MAC addresses.
- Int size
  Size of the entire packet (including any payload).

16.43 mon_setenv()

Similar to standard setenv().

PROTOTYPE:

void mon_setenv(char *varname, char *value);

DESCRIPTION:

The monitor can establish shell variables at the command line using the *set* command. This function allows an application to also establish a shell variable.

PARAMETERS:

- char *varname
  The name of the shell variable to be created.
- char *value
  The value that the shell variable represents. If this pointer is NULL, then the shell variable with the name *varname* is removed from the environment.

16.44 mon_setUserLevel()

Modify or retrieve the current user level of the monitor.

PROTOTYPE:

int mon_setUserLevel(int level, char *password);

DESCRIPTION:

Allows the application to query or modify the monitor's user level. At this API level, the password can be ignored. An incoming password of (char *)NULL tells this function not to check the password, but to simply adjust the user level. This allows the application to decide whether or not it actually wants to use the password protection of user levels in the monitor. Refer to section 5.6 above for more details on user levels within the monitor.

PARAMETERS:

- int level
  Value to be used as the new level. If -1, then simply return the current level. The current valid range for user levels in the monitor is 0-3.
- char *password
  The string that corresponds to the password needed to get to the specified user level. Note that (char *)NULL tells this function to ignore the password entry.

RETURN:

The current (if level == -1) or old user level.

16.45 mon_sprintf()

Similar to sprintf() but limited in the formatting capability.

PROTOTYPE:

int mon_sprintf(char *buffer, char *format,arg1, arg2, ...);

DESCRIPTION:

Provides the application with a small and simple sprintf() with limited formatting capability. The %s, %x, %c, and %d formats are supported to some degree. To keep it small and simple, no floating point conversion is supported. If %s points to a NULL, the string "NULL_POINTER" is printed. Also, non-standard format conversions for MAC and IP address formats are supported... %I assumes the argument is a long and it is converted to a string in IP format (1.2.3.4); %M assumes the argument is a pointer to an array of 6 bytes and it is converted to a string in the MAC address format (xx:xx:xx:xx:xx:xx).

PARAMETERS:

- char *buffer
  buffer into which the format conversion is to be placed
- char *format
  pointer to a format buffer
- argN
  arguments referred to (if any) by format buffer).

RETURN:

The size of the final string printed out the console port.

EXAMPLE use of %I & %M…

The following code…

```
int
main(int argc, char *argv[])
{
        static char mac[6] = { 0x11, 0x22, 0x33, 0x44, 0x55, 0x66 };
        long ip = 0x99410474;

        mon_printf("IP: %I, MAC: %M\n", ip, mac);
        return(0);
}
```

Generates this output…

```
IP: 153.65.4.116, MAC: 11:22:33:44:55:66
```

16.46 mon_timeofday()

Allows the application to set/retrieve/control the system's time-of-day clock (if present).

PROTOTYPE:

int mon_timeofday(int cmd, void *arg);

DESCRIPTION:

If the target system has a time-of-day clock and INCLUDE_TIMEOFDAY has been set to 1 in config.h, then this API may be used to set/retrieve/control the time-of-day clock. The function takes one of 4 commands:

- TOD_ON: enables the time-of-day clock (if applicable)
- TOD_OFF: disable the time-of-day clock (if applicable)
- TOD_SET: the incoming arg is a pointer to a struct todinfo used to set the time-of-day
- TOD_GET: the incoming arg is a pointer to a struct todinfo into which the current time of day is loaded.

PARAMETERS:

- int cmd
  current valid values are TOD_ON, TOD_OFF, TOD_SET, TOD_GET
- void *arg
  for TOD_GET and TOD_SET this is a pointer to struct todinfo (see date.h)

RETURN:

This function returns zero if successful; else negative.

16.47 mon_tfsadd()

Add a new file to TFS.

PROTOTYPE:

int mon_tfsadd(char *name, char *info, char *flags, unsigned char *src, int size);

DESCRIPTION:

The application can use this function to create a file from some block of memory without having to go through a typical open, write, close scenario. If the file already exists, it will first see if the incoming data is identical to that of the file already in TFS; if it is, then no flash operation is performed and TFS_OKAY is returned. If there are differences, then the new file is added and verified, then the old file is deleted.

PARAMETERS:

- char *name
  Name of the file being created.
- char *info
  Content to be placed in the info field of the header (or NULL).
- char *flags
  Flags to be assigned to the file (or NULL).
- unsigned char *src;
  Location of the data to become the file content.
- int size
  Size of the data copied to the file.

Warning:

Although it is allowed, it is dangerous to pass a source address into this function that is within TFS flash space. This would seem to be the intuitive thing to do to copy one file to another; however, if a defragmentation occurs as a result of this addition and the source file is shifted, then the add is aborted and the TFSERR_FLAKEYSOURCE error code is returned. Whenever copying from one file to another, it is best to copy the source file to local RAM space then transfer the content of the RAM to tfsadd(). Then, even if a defragmentation does occur, it will not affect anything in the source space.

RETURN:

TFS_OKAY if successful; else...
TFSERR_BADARG
TFSERR_CORRUPT
TFSERR_FILEEXISTS
TFSERR_FLASHFULL
TFSERR_FLASHFAILURE
TFSERR_BADCRC
TFSERR_FLAKEYSOURCE

COMPLETE LIST OF TFS RETURN CODES:
Note that all return error codes are negative except TFS_OKAY.

ERROR CODE RETURNED	Error Meaning
TFS_OKAY	No error
TFSERR_NOFILE	File not found
TFSERR_NOSLOT	Maximum number of files opened
TFSERR_EOF	End of file
TFSERR_BADARG	Bad argument
TFSERR_NOTEXEC	Not executable
TFSERR_BADCRC	Bad crc
TFSERR_FILEEXISTS	File already exists
TFSERR_FLASHFAILURE	Flash operation failed

TFSERR_WRITEMAX	Max write count exceeded
TFSERR_RDONLY	File is read-only
TFSERR_BADFD	Invalid descriptor
TFSERR_BADHDR	Bad binary executable header
TFSERR_CORRUPT	Corrupt file
TFSERR_MEMFAIL	Memory failure
TFSERR_NOTIPMOD	File is not in-place-modifiable
TFSERR_FLASHFULL	Out of flash space
TFSERR_USERDENIED	User level access denied
TFSERR_NAMETOOBIG	Name or info field too big
TFSERR_FILEINUSE	File in use
TFSERR_NOTCPRS	File is not compressed
TFSERR_NOTAVAILABLE	TFS facility not available
TFSERR_BADFLAG	Bad flag
TFSERR_CLEANOFF	Defragmentation is disabled
TFSERR_FLAKEYSOURCE	Dynamic source data
TFSERR_BADEXTENSION	Invalid file extension
TFSERR_LINKERROR	file link error
TFSERR_BADPREFIX	invalid device prefix
TFSERR_ALTINUSE	alternate devcfg in use
TFSERR_NORUNMONRC	can't run from monrc

16.48 mon_tfsclose()

Close a TFS file that had previously been opened.

PROTOTYPE:

int mon_tfsclose(int tfd, char *info);

DESCRIPTION:

When all interaction with an opened file is complete, mon_tfsclose() must be called to release the file descriptor used with the opened file and possibly initiate a transfer of the data to flash (if the file was opened for some type of modification). The *tfd* argument is the value that was returned from the initial mon_tfsopen(), and the *info* argument is a string (optionally NULL) that is used as the "*info*" field of the file header (if it is being modified or created).

NOTE: This is a significant difference between TFS and a standard open/close/read/write model for file IO. TFS does not actually write any data to flash until the file interaction is completed (i.e. when mon_tfsclose() is called).

PARAMETERS:

- int tfd
  The same value that was returned when the initial mon_tfsopen() was called.
- char *info
  A pointer to a string that is to be stored in the "info" field of the file header.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFS_OKAY if successful, else...
TFSERR_BADARG
TFSERR_BADFD
TFSERR_FLASHFAILURE

16.49 mon_tfsctrl()

Perform some type of control operation on TFS or a file in TFS.

PROTOTYPE:

int mon_tfsctrl(int rqst, long arg1, long arg2);

DESCRIPTION:

Similar in purpose to a standard ioctl() system call, this function allows the user to perform some type of control function on TFS or a file in TFS.

PARAMETERS:

- int rqst
  Type of control function to be performed.
- long arg1
  Depending on the value of rqst, this argument may or may not be used.
- long arg2
  Depending on the value of rqst this argument may or may not be used.

VALID RQST VALUES:

- TFS_CHECKDEV
  Arg1 is a char pointer to the name of the TFS device to be checked. If *arg1* is NULL, then all TFS devices are checked. Returns TFS_OKAY if file system on the specified device is not found to be corrupt.
- TFS_DEFRAG
  Run a TFS defragmentation, to remove any "dead" flash space taken up by deleted files.
  If *arg1* is non-zero, then after defragmentation, the target is reset; the value of *arg2* is considered the verbosity level to use during defragmentation.
- TFS_DEFRAGDEV
  Arg1 is a char pointer to the name of the TFS device to be defragmented (cleaned). Returns TFS_OKAY if successful.
- TFS_DOCOMMAND
  Arg1 is a pointer to the application's command interpreter function whose prototype is *void docommand(char * cmdline, int verbosity).* If NULL, then the standard command interpreter is used.
  Arg2 is a pointer to a location into which the monitor will place the current command interpreter function being used by the script runner. If NULL, then this value is not loaded. Returns TFS_OKAY if successful. Refer to section 7.13 for more details on this.
- TFS_ERRMSG
  Returns a pointer to a character string that corresponds to the verbose description of the error.
  The value in *arg1* is some error value that was returned by some other TFS system call.
- TFS_FATOB
  This request converts a string of flag characters (see TFS attributes in section 5.1) to a binary value that is stored in the file header. *Arg1* contains a pointer to the string of characters. The return value is a long that represents the binary value used by TFS internally. The return value is -1 if any character of the incoming string is invalid.
- TFS_FBTOA
  Just the opposite of TFS_FATOB... It takes a binary value and converts it to an ASCII string. *Arg1* is the binary value, *arg2* is a pointer to the buffer (should be at least 16 bytes) into which TFS stores the string.
- TFS_FCOUNT
  Return the number of files in TFS or within one device within TFS space. The value of *arg1* (if non-zero) is assumed to be the name of the TFS device; if NULL, then a count of all files (regardless of device) is returned.
- TFS_HEADROOM
  Based on the offset into the file specified by the incoming file descriptor (arg1), return the gap between the current offset and the end of the file.
- TFS_INITDEV
  Arg1 is a char pointer to the name of the TFS device to be initialized. Returns TFS_OKAY if successful.

- TFS_MEMAVAIL
  Returns the amount of flash memory that is still available for use by TFS.
- TFS_MEMDEAD
  Returns the amount of flash memory that is currently being used by deleted files.
- TFS_MEMUSE
  Returns the amount of flash memory that is currently being used by files in TFS. This includes space used by active and deleted files.
- TFS_RAMDEV
  Arg1 is TRAMDEV pointer (see tfs.h) which must contain the new RAM device configuration. Returns TFS_OKAY if successful.
- TFS_TELL
  Return the offset into the file specified by *arg1* which is the file descriptor returned by mon_tfsopen() sometime prior.
- TFS_TIMEFUNCS
  Arg1 is a pointer to the getLtime() function and *arg2* is a pointer to the getAtime() function. These two functions are target specific and this is only applicable if the target has a battery backed time of day clock. Refer to discussion below. Returns TFS_OKAY if successful.
- TFS_UNOPEN
  If a TFS file was previously opened for creation or append, and for some reason, the need to create/modify the file no longer exists, this function essentially calls mon_tfsclose() but does not make any modifications to the flash. The value of *arg1* is the file descriptor returned by the initial call to mon_tfsopen().

If in the above descriptions, one or both of the *'arg'* values is not mentioned, then assume it is not used.

DISCUSSION: TFS & Time-of-File-Creation

Since the basic model of the monitor is to run without the need of any interrupts from the host processor, how is it that TFS can store the time at which a file is created? It depends on the target-specific code to provide it with two functions that will support this: *getLtime()* and *getAtime()*. The function (long)getLtime(void), must return a long that is stored in the header of the TFS file when it is created. The function (char *)getAtime((long *)tval, (char *)buf, (int)buflen) can be used to simply return an ASCII string representing the current time (if tval is 0) or it can return an ASCII string representing the value stored in tval. The value in tval will typically be the value that was previously returned from *getLtime()*. With this interface, TFS really doesn't have a clue about time-of-day, but it uses the capabilities given to it by the target-specific code to make it look like it does.

Note that this is a feature that is used by TFS to populate an entry in the header of the file being written at the time. If the two above functions are not supplied to TFS, then the header entry is left blank, and the file simply has no recollection of its time of creation. These functions may be provided to TFS by target-specific code in the monitor itself, of through the use of the tfsctrl(TFS_TIMEFUNCS,…) function mentioned above.

DISCUSSION: Multiple TFS Storage Devices

In some hardware designs there may be more than one device that could be used for file storage. TFS supports this. A basic system will have a boot monitor in the base of the flash, then all remaining flash in that device is used by TFS and that's it. A more complicated system may contain battery backed ram, a boot flash device and a secondary storage flash (or ram) device, etc... TFS supports multiple devices that are not necessarily in contiguous address space. Each device appears to the user as a directory, so any file can be stored in any device (limited by the size of the device, of course), but a file cannot span across multiple non-contiguous devices. For each device, the same power-safe defragmentation method is used; hence, if battery-backed RAM was on-board, it could be used to eliminate the problem of flash-life expectancy (see below) if there is a need to modify files at a high frequency.

To "steer" a file to a particular device, each device has a unique prefix that, when made part of the file name, tells TFS that the file is destined for that device. If the prefix is omitted from the filename, then the default device (whichever device is at the top of tfsdevtbl[] in tfsdev.h) is used for storage. Similarly, the file system maintenance commands (tfs check, tfs clean, tfs freemem, etc...) can also be pointed to a particular device by specifying the device prefix.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):
The value returned depends on the 'rqst' value passed into the function, see above.

EXAMPLE use of mon_tfsctrl() for returning an error string:

The following code …

```
int
main(int argc,char *argv[])
{
  int tfd;
  char *file;

  file = "test";
  tfd = mon_tfsopen(file,TFS_RDONLY,0);
  if (tfd < 0)
    mon_printf("%s: %s\n", file,(char *)mon_tfsctrl(TFS_ERRMSG,tfd,0));
  return(0);
}
```

generates output as follows (assuming the file “test” does not exist)…

```
test: file not found
```

16.50 mon_tfseof

Return EOF (end of file) status on specified file.

PROTOTYPE:

int mon_tfseof(int tfd);

DESCRIPTION:

Allows the application to check to see if a currently opened-for-read file has reached the end-of-file.

PARAMETERS:

- int tfd

  TFS file descriptor returned from a previous call to mon_tfsopen().

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

1 if TFS's internal pointer has reached the end of the file; 0 if not at the end of file; else negative indicating some error: TFSERR_BADARG, TFSERR_BADFD.

16.51 mon_tfsfstat()

Populate a TFILE structure with the designated file's file header structure.

PROTOTYPE:

int mon_tfsfstat(char *filename, TFILE *tfsstruct);

DESCRIPTION:

Allows the application to retrieve a TFILE structure (struct tfshdr) attached to the specified file (if it exists). This API function is a replacement for tfsstat() in cases where the pointer returned by tfsstat() is used for more than just determining file existence.

PARAMETERS:

- char *filename
  name of file in TFS.
- TFILE *tfsstruct
  pointer to a TFILE structure that mon_tfsfstat will populate if the file exists.

RETURN:

Return 0 if the file exists; else -1.

TFILE (struct tfshdr):

```
/* struct tfshdr:
 *     It is in FLASH as part of the file system to record the attributes of
 *     the file at the time of creation.
 */
struct tfshdr {
       unsigned short hdrsize;     /* Size of this header.     */
       unsigned short hdrvrsn;     /* Header version #.        */
       long filsize;               /* Size of the file.        */
       long flags;                       /* Flags describing file.   */
       unsigned long filcrc;             /* 32 bit CRC of file.      */
       unsigned long hdrcrc;             /* 32 bit CRC of the header.*/
       unsigned long modtime;            /* Time when file was last modified.      */
       struct tfshdr *next;              /* Pointer to next file in list.          */
       char name[TFSNAMESIZE+1];         /* Name of file.                          */
       char info[TFSINFOSIZE+1];         /* Miscellaneous info field.              */
#if TFS_RESERVED
       unsigned long rsvd[TFS_RESERVED];
#endif
};
```

16.52 mon_tfsgetline()

Retrieve the next line from an assumed ASCII file.

PROTOTYPE:

int mon_tfsgetline(int tfd, char *buffer, int max);

DESCRIPTION:

This function retrieves the next line of characters from an opened ASCII-readable file. Retrieval continues until either 'max-1' characters are loaded or a CR and/or LF is found. If max-1 chars are loaded, the final character placed in the buffer is a NULL. If CR and/or LF is loaded, it is followed by a NULL. The returned buffer will always be NULL terminated.

PARAMETERS:

- int tfd
  The descriptor of the file, returned previously by mon_tfsopen().
- char *buffer
  A pointer to the space into which TFS is to place the specified number of bytes.
- int max
  The max number of bytes to place into the buffer.

RETURN:

The number of bytes retrieved if successful, else the error returned from tfsread().

16.53 mon_tfsinit()

Initialize the flash space that is used by TFS.

PROTOTYPE:

int mon_tfsinit(void);

DESCRIPTION:

All of the flash space is erased by this function, so **be careful**!

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

Returns TFS_OKAY if successful; else negative indicating a flash operation error (should not happen).

16.54 mon_tfsipmod()

Make a modification to a file directly in the flash space that it currently resides in.

PROTOTYPE:

int mon_tfsipmod(char *name, char *buffer, int offset, int size);

DESCRIPTION:

This function is unique to TFS. It provides the user with the ability to modify an existing file without deleting the old one then writing a new one. This requires some knowledge of the underlying flash. Note that this method of access does not require the typical open(), read/write, close() calls to modify the flash.

DISCUSSION:

Typically, when a file in TFS is modified, the original file is marked as deleted, and the new version of the file is appended to the end of the list of files currently stored in flash. This can involve a relatively large amount of overhead if the modification to be made is trivial. As an alternative, a file can be created as an "in-place-modifiable" file which means that the API provides a means by which a file can be modified without the typical deletion/re-creation step mentioned above. This is done by creating the file as in-place-modifiable and specifying the file to be of some size. The space is then allocated in TFS for this file, but the flash is all left in a writeable state. This usually means that the bytes in the flash are all 0xff (usually, bits in flash can be cleared on a byte-by-byte basis, but to reset them, an entire sector is affected). All subsequent writes to this file, then, are done directly to the currently allocated flash instead of to a new block of flash. Obviously this puts some responsibility back on the programmer, but it can potentially save quite a bit of overhead if necessary. When a file is created as in-place-modifiable, the TFS API function mon_tfsipmod() should be used instead of the standard open-modify-close model.

PARAMETERS:

- char *name
  Name of the file to be modified
- char *buffer
  New data to be written to flash.
- int offset
  Offset into file into which new data is to be written. If this value is -1, then TFS will automatically use the first location that contains 0xff.
- int size
  The number of bytes for TFS to write.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFS_OKAY if successful, else...
TFSERR_NOFILE
TFSERR_NOTIPMOD
TFSERR_BADARG
TFSERR_WRITEMAX
TFSERR_FLASHFAILURE

16.55 mon_tfslink()

Link a new filename (target) to an existing file (source).

PROTOTYPE:

int mon_tfslink(char *source, char *target);

DESCRIPTION:

Allows the application to establish a "linkage" between multiple files in TFS.

PARAMETERS:

- char *source
  Name of the file already in TFS space.
- char *target
  Name of new file that will be the linkfile.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFS_OKAY if successful, else...
TFSERR_NOFILE
TFSERR_USERDENIED
TFSERR_FLASHFAILURE

NOTES:

The idea of a link in TFS is simply the ability to use one file name (the link) to access some other file indirectly. The only case where this is not applicable is file removal (tfsunlink() at the API and "tfs rm" at the CLI). In this case, the immediate file name, whether it be a link or not, is removed.

WARNING:

If a file (target) is a link to another file (source), and at some point the source file is removed, the link file will still exist. This means that the link file will be redirected to a file that is non-existent; hence it will take on the same attribute.

16.56 mon_tfsnext()

Go to next header pointer in TFS flash space.

PROTOTYPE:

struct tfshdr *mon_tfsnext(struct tfshdr *tfp);

DESCRIPTION:

Allows the application to step through the list of files in TFS..

PARAMETERS:

- struct tfshdr *tfp
  Pointer to a TFS file header to be used to retrieve the next header. If this pointer is NULL, then a pointer to the header of the first valid file in TFS is returned.

RETURN:

A pointer to the header of the next valid file after the header passed as a parameter or (struct tfshdr *)NULL if no more headers.

EXAMPLES:

The following 2 functions are examples of usage of mon_tfsnext() to list files stored in TFS. The first function (tfsls) is a basic listing. The second function (tfsvls) is a listing with header information also displayed to the user…

```
void
tfsls(void)
{
    TFILE *tfp;

    tfp = (TFILE *)0;

    while((tfp = mon_tfsnext(tfp))) {
        mon_printf(" %s\n",TFS_NAME(tfp));
    }
}

int
tfsvls(void)
{
    int tot;
    char flags[16];
    TFILE *tfp;

    tot = 0;
    tfp = (TFILE *)0;

    mon_printf(" Name                    Size    Location  Flags  Info\n");
    while((tfp = mon_tfsnext(tfp))) {
        tot++;
        mon_tfsctrl(TFS_FBTOA,TFS_FLAGS(tfp),(long)flags);
        mon_printf(" %-20s  %7d  0x%08x  %-5s  %s\n",
            TFS_NAME(tfp),TFS_SIZE(tfp),
            TFS_BASE(tfp),*flags != 0 ? flags: " ",TFS_INFO(tfp));
    }
    return(tot);
}
```

16.57 mon_tfsopen()

Open up a TFS file for read and/or write access.

PROTOTYPE:

int mon_tfsopen(char *filename,long flagmode, char *buffer);

DESCRIPTION:

Similar to a standard open() of a file, this function allows the user to open a TFS file for access. The final buffer argument is needed only for files that are to be created or modified. This is the space that is used by TFS for building the file. As multiple mon_tfswrite() calls are made, the data written is placed in this buffer; then when mon_tfsclose() is called to complete the file transaction, the buffer is transferred to flash to become a permanent part of the file system. Note that the final buffer argument should be NULL if the file is opened for read-only.

PARAMETERS:

- char *filename
  Name of the file to be read, written or created.
- long flagmode
  The flags to be applied to the file when closed and the mode that the file is to be opened with.
- char *buffer
  A pointer to memory space that will be used by TFS while the file is being generated.

Valid modes:

- TFS_RDONLY: file is assumed to already exist, and it is being opened for read only.
- TFS_APPEND: file is assumed to already exist, and it is being opened to append to the end of the current file. If the file does not exist, then an error (TFSERR_NOFILE) is returned.
- TFS_CREATE: file is assumed to not exist, and it is being created. If the file does exist, an error (TFSERR_FILEEXISTS) is returned.

In general, only one mode should be specified. An exception to this is TFS_APPEND|TFS_CREATE. If both of these modes are specified, then TFS will modify the mode based on the presence of the file... If the file exists, then it is opened with TFS_APPEND; if the file doesn't exist, it is opened with TFS_CREATE.

Valid flags (these flags correspond to the content of the table in section 5.1):

- TFS_EXEC: executable
- TFS_BRUN: to be automatically executed at boot time
- TFS_QRYBRUN: to be automatically executed at boot time, after querying at the console.
- TFS_COFF: loadable executable is COFF format.
- TFS_ELF: loadable executable is ELF format.
- TFS_AOUT: loadable executable is AOUT format.
- TFS_CPRS: file is compressed.
- TFS_UNREAD: file cannot even be read at a user level lower than its own.
- TFS_ULVLN: file is accessible by user level N and above, where N can be 0-3.
- In general, multiple flags are specified for a file. For example...
- TFS_EXEC | TFS_QRYBRUN | TFS_ELF | TFS_ULVL2: is a valid flag specification indicating that the file is executable ELF that will autoboot with query and will only be executable by user levels greater than or equal to 2.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

Any number greater than or equal to zero if successful, else…

TFSERR_NOFILE
TFSERR_USERDENIED
TFSERR_FILEEXISTS
TFSERR_BADARG

TFSERR_MEMFAIL
TFSERR_NOSLOT

16.58 mon_tfsread()

Access a file that has been previously opened for reading and retrieve data from that flash space.

PROTOTYPE:

int mon_tfsread(int tfd, char *buffer, int size);

DESCRIPTION:

Similar to a standard read() of a file, this function allows the user to retrieve data from a file that has been previously opened.

PARAMETERS:

- int tfd
  The descriptor of the file, returned previously by mon_tfsopen().
- char *buffer
  A pointer to the space into which TFS is to place the specified number of bytes.
- int size
  The number of bytes to place into the buffer.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

The number of bytes retrieved if successful, else negative...
TFSERR_BADARG
TFSERR_BADFD
TFSERR_EOF
TFSERR_MEMFAIL

16.59 mon_tfsrun()

Execute a file in TFS.

PROTOTYPE:

int mon_tfsrun(char *arglist[], int verbose);

DESCRIPTION:

Allows the application to execute a file stored in TFS flash space.

PARAMETERS:

- char *arglist[]
  An argument list (similar to argv[] passed into main()) with a null pointer after the last entry.
- int verbose
  Verbosity level (0, 1 or 2)...
  0: no verbosity
  1: print list of arguments for each command in script after tokenization
  2: print list of arguments showing value in shell variables.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFS_OKAY if successful, else...
TFSERR_NOFILE
TFSERR_USERDENIED
TFSERR_NOTEXEC
TFSERR_BADCRC

16.60 mon_tfsseek()

Move the internal pointer maintained by TFS to some specified position.

PROTOTYPE:

int mon_tfsseek(int tfd, int offset, int whence);

DESCRIPTION:

Similar to a standard lseek() of a file, this function allows the user to adjust the current pointer maintained by TFS for the specified file.

PARAMETERS:

- int tfd
  Descriptor of the file whose pointer is to be adjusted.
- int offset
  Offset relative to location specified by 'whence'
- int whence
  Base position from which the offset is assumed.

Valid values for whence:

- TFS_BEGIN: specified offset is relative to the beginning of the file.
- TFS_CURRENT: specified offset is relative to the current position in the file.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

The offset into the file if successful, else negative…
TFSERR_BADARG
TFSERR_EOF

16.61 mon_tfsstat()

Return a TFILE pointer to the file specified.

PROTOTYPE:

struct tfshdr *mon_tfsstat(char *filename);

DESCRIPTION:

Allows the application to retrieve a TFILE pointer (struct tfshdr *) to the specified file (if it exists).

PARAMETERS:

- char *filename
  Name of file in TFS.

RETURN:

A pointer to the header of the specified file or (struct tfshdr *)NULL.

WARNING: if this system call is used in an application that will be automatically defragmenting occasionally, AND the pointer returned by tfsstat() is used for something more than just determining if the file exists, then if possible, use tfsfstat() instead.

16.62 mon_tfstell()

Return the current offset into the file referred to by the incoming descriptor.

PROTOTYPE:

int mon_tfstell(int tfd);

DESCRIPTION:

Allows the application to determine current offset into the specified file.

PARAMETERS:

- int tfd
  Descriptor (returned by mon_tfsopen()) of the file

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFSERR_BADARG if failure; else the offset.

16.63 mon_tfstruncate()

Truncate the size of a file that has been opened for append to a new size.

PROTOTYPE:

int mon_tfstruncate(int tfd, int size);

DESCRIPTION:

If a file is opened for writing (TFS_APPEND flag passed to tfsopen), and the modifications to this file require that the new file size be smaller, then the file size must be truncated. This function provides that capability. In early versions of TFS, when a file was opened for modification, when it was closed it was closed with a size that was dependent on the current position of the write pointer. This was wrong, and has been fixed as of Sept 2000 TFS. Code that originally depended on this TFS bug must now use mon_tfstruncate() prior to doing the call to tfsclose().

PARAMETERS:

- int tfd
  The descriptor of the file, returned previously by mon_tfsopen().
- int size
  The new, smaller size of the file.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFS_OKAY if successful else...
TFSERR_BADARG
TFSERR_BADFD

16.64 mon_tfsunlink()

Remove a file from TFS flash space.

PROTOTYPE:

int mon_tfsunlink(char *filename);

DESCRIPTION:

Allows the application to remove a file from TFS flash space.

PARAMETERS:

- char *filename
  Name of the file to be removed.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFS_OKAY if successful, else...
TFSERR_NOFILE
TFSERR_USERDENIED
TFSERR_FLASHFAILURE

16.65 mon_tfswrite()

Access a file that has been previously opened for writing and transfer data to TFS for eventual transfer to flash.

PROTOTYPE:

int mon_tfswrite(int tfd, char *buffer, int size);

DESCRIPTION:

Similar to a standard write() of a file, this function allows the user to place data into a file that was previously opened for writing.

PARAMETERS:

- int tfd
  The descriptor of the file, returned previously by mon_tfsopen().
- char *buffer
  A pointer to the space from which TFS is to copy the specified number of bytes.
- int size
  The number of bytes for TFS to copy from the buffer.

RETURN (see mon_tfsadd() for a complete list & description of the TFS return codes):

TFS_OKAY if successful else...
TFSERR_BADARG
TFSERR_RDONLY
TFSERR_MEMFAIL

16.66 mon_version()

A mechanism by which the application can retrieve the monitor's version information. As of uMon1.0, the monitor version is a 3-digit, dot-delimited number: X.Y.Z where “X.Y” is the major/minor version of the core uMon1.0 source code (common to all targets) and “.Z” is the version of the target port. Note that the version information and the build-date of the monitor are also available through the shell variables VERSION_MAJ, VERSION_MIN, VERSION_TGT and MONITORBUILT. Refer to the description of each of these shell variables for more information.

PROTOTYPE:

```
char * mon_version(void);
```

DESCRIPTION:

This command returns a null terminated string containing the “X.Y.Z” value discussed above.

16.67 mon_watchdog()

Allows the application call the same watchdog function that is used by the monitor.

PROTOTYPE:

```
#include "monlib.h"
int mon_watchdog(void);
```

DESCRIPTION:

For systems that have a watchdog that needs to be periodically tickled, the monitor has a WATCHDOG_MACRO that can be established and will be called periodically to do whatever it has to do to keep the watchdog timer from expiring. That same watchdog mechanism must be run when the application takes over, so this API eliminates the need to duplicate the code in the application's space.

In many cases the hardware doesn't have a watchdog macro because there is no watchdog hardware installed on the system. The application can check the return value from mon_watchdog() and if it returns –1, this is indication that no WATCHDOG_MACRO is installed in the monitor; hence, there is probably no need to run a watchdog in the application.

16.68 mon_warmstart()

Allows the application to re-initialize various portions of the monitor without the need to actually reset the target.

PROTOTYPE:

```
#include "monlib.h"
void mon_warmstart(unsigned long options);
```

DESCRIPTION:

Typically the target system will boot up in sequence... After reset, the monitor will boot the basic target, then eventually the application is launched out of TFS to start doing whatever the target is really supposed to do. The application then has access to various monitor-initialized facilities. There are cases where the application may startup and the monitor is never invoked. This is the situation when an application is being downloaded to the target through some type of external debugger (BDM or JTAG for example). The debugger knows of the application, not the monitor; but the application still wants to be able to use some of the monitor hooks. The problem here is that the application has been remotely started through the debugger and now wants to use facilities in the boot monitor that have not been initialized because the target never went through its natural reset process.

This hook solves this problem. It provides the application with the ability to "warmstart" portions of the monitor that are accessible by the application. Various options are available...

- **WARMSTART_IOINIT:**
  Initialize the monitor-owned IO (Ethernet & serial usually)
- **WARMSTART_BSSINIT:**
  Initialize monitor-owned bss space.
  Warning: if this flag is set, then the calling application MUST already be running from its own pre-defined stack space. This is because when the monitor starts up an application, the application defaults to using the monitor's stack; however, the monitor's stack is within it's own .bss space. The result is that if a pre-defined stack was not set up prior to this point, this call will clear the stack causing unknown (but certainly deadly) results.
- **WARMSTART_RUNMONRC:**
  Run through the execution of the monrc file.
- **WARMSTART_MONHEADER:**
  Dump the monitor header to the console
- **WARMSTART_TFSAUTOBOOT:**
  Run through the TFS autoboot sequence.
- **WARMSTART_ALL:**
  Initialize all of the above.

Chapter 17 Host-Based Tools

Micromonitor has been around for a while. It was being used prior to the prevalence of the GNU cross-compilation tools now taken for granted in the industry. As a result, there was a need for some host based tools. Some were written years ago and have since become less useful because of various other options that have become available to the development community (i.e. GNU). Some are used during the MicroMonitor build process and others are applicable only at runtime. The majority of these tools have already been discussed and/or used in some detail in previous sections of this text. The purpose of this section is to provide a quick reference to each of the tools and their options.

17.1 Building the Tools

As of uMon 1.0, the majority of the source code for the tools that are used with MicroMonitor are included in the package and build using native GNU-GCC. The umon_main/host directory is the top level directory under which all of the tools are built and installed. By default, the tools are installed in umon_main/host/bin; and the makefiles used in the port-specific directory refer to this directory. It may be more convenient to copy the installed toolset into whatever PATH is best suited for the system you are working on.

The first thing to notice is the umon_main/host/README file. Always assume that the information in that README is more up-to-date than this document, so whenever there are conflicts assue that the README is the authority. The README sums it all up into one statement…

```
make OSTYPE=cygwin|linux|solaris rebuild
```

This command will build and install all the tools to umon_main/host/bin. Note that the available port directories assume that these tools are available, so it's important that these tools be built and installed prior to attempting a port build. If for some reason you would prefer that they be installed somewhere else, then simply add BIN=DIR_PATH where DIR_PATH is the name of the directory you want them to install to.

17.2 Building Tools with Visual C++

By default, the host based tools build using native GCC and this is fine as long as you have some version of GNU tools on your host system. To build the tools using Visual C++, add the declaration VCC=TRUE to the make line…

```
make VCC=TRUE OSTYPE=cygwin|linux|solaris rebuild
```

17.3 AOUT

Extract information from or compress sections of an AOUT file.

USAGE:

aout [-a:AB:cfMm:sS:v] {aout_filename}

DESCRIPTION:

This tool is one of three tools (coff, elf & aout) that basically perform the same tasks, but on different file formats. It allows the user to dump different portions of the file header to stdout in a readable format. Also, future plans include support for using zlib to compress each of the .text and .data sections so that TFS can store a compressed file in flash and decompress directly from TFS to the absolute-linked location for the application to run in.

OPTIONS:

- -a {filename}
  append file to end of -S file;
- -A
  print what was previously appended by -a
- -B {bin_filename}
  convert file (aout_filename) to binary (bin_filename) for transfer to absolute memory space;
- -c
  BE-to-LE convert... By default, this tool assumes the aout file was built with big-endian control structures. This option will convert to little endian control structures.
- -f
  show AOUT file header;
- -M
  show AOUT map with file offsets;
- -m
  show AOUT map without file offsets;
- -p {filename}
  pack to specified file (output file is also stripped)
  This is soon to be replaced with zlib compression.
- -s
  show AOUT section headers;
- -S {filename}
  strip to specified file
- -v
  verbose (debug) mode

NOTES:

- If, after running this tool, some type of file error (read, write, lseek, etc...) occurs, it is likely that the executable being operated on was built with control structures of a different endian-type. Try re-running with the -c option.

EXAMPLES:

- aout -a info -S app.str app
  Strip the file *app*, place the result in *app.str*, then append the content of the file *info* to the end of *app.str*. The command line "strings app.str | tail -1" can then be run to extract the last string (content of *info* file) from the binary.

EXIT STATUS:

0 if successful
1 ERROR

17.4 BIN2ARRAY

Convert a binary file to compile-able array in 'C'.

USAGE:

bin2array [-Aa:b:D:e:no:sw:V] {input_file} [array-name]

DESCRIPTION:

This tool takes a binary file and converts it to an ASCII file that contains a compile-able 'C' array that contains the ASCII-coded hex equivalent of the content of the file. For example, if I have a file with the following content...

```
1234567890
```

Then bin2array would create something like this...

```
unsigned char some_array[] = {
    0x31, 0x32, 0x33, 0x34, 0x35, 0x36, 0x37, 0x38, 0x39, 0x30, 0x0a
};
```

Notice that the 0x31, 0x32, 0x33, etc...(from the output file) is the ASCII-coded-hex equivalent of 123 etc... (from the input file).

OPTIONS:

- -A
  use assembler byte format
- -a {arrayname}
  specify the array name to be used. If not specified, then the array name will match the filename.
- -b {offset}
  offset into binary file to begin at
- -D {define}
  up to 16 "#define" statements inserted in output
- -e {offset}
  offset into binary file to end at
- -n
  null-terminate the created array.
- -o {filename}
  output filename (default = stdout)
- -s
  swap data
- -w {width}
  unit width (1|2|4) (default = 1)
- -V
  display version (build date) of tool

EXAMPLES:

- bin2array -a file_bin -o file_array file
  Convert the entire content of 'file' in the array "file_bin[]" stored in file "file_array.c", .

EXIT STATUS:

0 if successful, else 1

17.5 BIN2SREC

Convert a binary file to an S3 record file.

USAGE:

bin2srec [-b:o:] {input_file}

DESCRIPTION:

This tool takes a binary file and converts it to S3-record format. The incoming file can be converted to exist at some address other than zero by specifying a base address; also, an offset into the input-file can be specified to adjust the starting point of the conversion.

OPTIONS:

- -b {base address}
  override default base address of 0;
- -o {offset}
  specify an offset (default is none) into the input_file at which point data conversion is started;

EXAMPLES:

- bin2srec file
  Convert the entire content of 'file' to S-Record format.

EXIT STATUS:

0 if successful, else 1

17.6 COFF

Extract information from or compress sections of a COFF file.

USAGE:

coff [-a:AB:cfMmp:sS:v] {coff_filename}

DESCRIPTION:

This tool is one of three tools (coff, elf & aout) that basically perform the same tasks, but on different file formats. It allows the user to dump different portions of the file header to stdout in a readable format. Also, future plans include support for using zlib to compress each of the .text and .data sections so that TFS can store a compressed file in flash and decompress directly from TFS to the absolute-linked location for the application to run in.

OPTIONS:

- -a {filename}
  append file to end of -S file;
- -A
  print what was previously appended by -a
- -B {bin_filename}
  convert file (coff_filename) to binary (bin_filename) for transfer to absolute memory space;
- -c
  BE-to-LE convert... By default, this tool assumes the coff file was built with big-endian control structures. This option will convert to little endian control structures.
- -f
  show COFF file header;
- -M
  show COFF map with file offsets;
- -m
  show COFF map without file offsets;
- -p {filename}
  pack to specified file (output file is also stripped)
  This is soon to be replaced with zlib compression.
- -s
  show COFF section headers;
- -S {filename}
  strip to specified file
- -v
  verbose (debug) mode
- -V
  print the time/date the tool was built
- -z {zip level}
  compress the COFF sections using zlib compression level "zip level". Refer to TFS decompression for more details on this.

NOTES:

- If, after running this tool, some type of file error (read, write, lseek, etc...) occurs, it is likely that the executable being operated on was built with control structures of a different endian-type. Try re-running with the -c option.

EXAMPLES:

- coff -a info -S app.str app
  Strip the file *app*, place the result in *app.str*, then append the content of the file *info* to the end of *app.str*. The command line "strings app.str | tail -1" can then be run to extract the last string (content of *info* file) from the binary.

EXIT STATUS:

0 if successful, else 1

17.7 DEFDATE

Simple utility to generate the text to build a header file containing current time & date.

USAGE:

defdate [-fnV] [macro_name]

DESCRIPTION:

This tool provides an alternative to a compiler's __DATE__ and __TIME__ macros. The intent is that this be used within a makefile to update the content of a header file whose macro is used somewhere in source code to keep track of the time at which the code was built. If an argument is present, defdate will print out a string that can be part of a header file (#define MACRO "datestring") ; if no argument is present, then only the date string itself is printed.

OPTIONS:

- -f {format}
  override the default format of %b_%d,%Y@%H_%M...
  - %a: abbreviated weekday name
  - %A: full weekday name
  - %b: abbreviated month name
  - %B: full month name
  - %c: date and time representation appropriate for locale
  - %d: day of month as decimal number (01-31)
  - %H: hour in 24-hour format (00-23)
  - %I: hour in 12-hour format (01-12)
  - %j: day of year as decimal number (001-366)
  - %m: month as decimal number (01-12)
  - %M: minute as decimal number (00-59)
  - %p: current locale's AM/PM indicator for 12-hour clock
  - %S: second as decimal number (00-59)
  - %U: week of year as decimal number, with Sunday as first day of week (00-51).
  - %w: weekday as decimal number (0-6; Sunday=0)
  - %W: week of year as decimal number, with Monday as first day of week (00-51).
  - %x: date representation of current locale
  - %X: time representation for current locale
  - %y: year without century, as decimal number (00-99)
  - %Y: year with century, as decimal number
  - %z: time-zone name or abbreviation; no chars if time zone is unknown
  - %Z: all-caps version of %z
- -n don't append a newline character to the end of the date string;
- -V print the version of defdate.exe

EXAMPLES:

- defdate DATETIME info.h
  Create a header file with a string formatted as follows: #define DATETIME "May_21,1999@1159"
- defdate
  Print the current date in the format May_21,1999@1159.

EXIT STATUS:

0 if successful
1 if error

17.8 DHCPSRVR

Run a basic dhcp or bootp server.

USAGE:

dhcpsrvr [-A:a:bDdc:Chq:TvVw] {target ip-address} "target command string"

DESCRIPTION:

This tool provides a basic DHCP or BOOTP server for MicroMonitor clients. For DHCP, 'automatic allocation' mode is supported (no lease expiration). The intent of the program is STRICTLY for basic support of MicroMonitor clients. It expects to interact with only one client at a time, and simply provides a convenient alternative to setting up a "real" server somewhere. The server requires a configuration file to startup. It can be automatically generated via *dhcpsrvr -C.* Refer to notes in that output for syntax information.

Complete documentation on the dhcpsrvr tool can be dumped to a file via *dhcpsrvr -h*. Where differences are detected (between that output and this text), assume dhcpsrvr -h output is more accurate.

OPTIONS:

- -A {arpcmdname}
  self-extract and build an arp command for use by this server if not available on the system already;
- -a {arpcmdname}
  use the specified string as the command name replacement for "arp";
- -b
  broadcast reply to client (eliminates need for arp);
- -C
  dump a template config file to stdout;
- -c {cfgfile_name}
  override the default (dhcpsrvr.cfg) config file name;
- -D
  don't issue the arp;
- -d
  don't check for the existence of an arp entry, just execute arp -d regardless;;
- -h
  dump more help information to stdout;
- -q {n}
  quit after 'n' discover/request handshakes;
- -T
  start up a TFTP server (recommendation: use *ttftp srvr* because this is soon to be eliminated from dhcpsrvr*);*
- -v
  verbose mode;
- -V
  print version of dhcpsrvr tool (build date);
- -w
  don't print warnings of incomplete .cfg entries;

EXAMPLES:

- dhcpsrvr
  Based on the file "dhcpsrvr.cfg" in the current working directory; start up a dhcp/bootp server.
- dhcpsrvr -C dhcpsrvr.cfg
  Create a template dhcpsrvr config file.
- dhcpsrvr -A myarp; dhcpsrvr -a myarp
  Build an "arp" tool called myarp and then tell the dhcp server to run with that version of arp.

WARNING:

When a DHCP server is to respond to a client that is making a request it has a catch-22 to deal with... The server's TCP/IP stack wants to talk to the client through the clients IP address, but the client doesn't have

an IP address assigned to it yet (that's what the server is trying to give it). The server has two alternatives: one is to force an entry into the server's ARP cache (using the arp command mentioned above); the other is to broadcast the response. Ideally, the arp-cache modification is used because it reduces traffic on the network. This is the default mechanism used by this server. Users of this dhcp server may not have Administrator privileges on the machine (required to do an arp cache modification), so the broadcast method can be used (see -D & -b options above). On most NT/Win95 systems, if the server is not able to modify the arp cache, an error is returned when the attempt is made. The user sees this error message and knows that the server must be reconfigured for broadcast. Some users have found that on some NT/Win95 boxes the arp fails but there is no warning. Be aware of this, it appears to be an inconsistency across various Window's platforms. The resolution is to just use broadcast mode (-Db options) so that the server is not dependent on the arp cache entry being made.

EXIT STATUS:
0 if successful
1 if error

EXAMPLE CONFIG FILE (output of dhcpsrvr -C):

```
# Configuration File Template....
# Notes:
# * Each line consists of an ID string followed by a PARAMETER string.
#   ID and PARAMETER must be white space delimited.
# * Blank lines are ignored, and lines starting with a '#' are ignored.
# * Specifying a client MAC address of 00:00:00:00:00:00 tells the server
#   to use this as a default.  Note that this should be the last entry in
#   the config file because the server will scan this file from top to
#   bottom searching for a matching CLIENT_MAC entry, if a match is not
#   found by the time this entry is scanned, the default will be used.
# * A complete entry begins with the CLIENT_MAC ID.  The server expects to
#   find all other entries associated with that MAC prior to finding the
#   next CLIENT_MAC ID.
# * The server will respond to BOOTP requests also.  To signify
#   a BOOTP (instead of a DHCP) entry, use BOOTP_CLIENT_MAC instead of.
#   DHCP_CLIENT_MAC.
# * The subnet of the PC that is running this server must be the same
#   subnet that the CLIENT_IP entries are set to.
# * The XXX_OPTNO_NNN entries below demonstrate the fact that the server
#   can be told to load any option with a hex, ASCII or IP type of value.
# * The XXX_VSOPTNO_NNN entries below demonstrate the fact that the server
#   can be told to load any vendor-specific option (#43) with a hex, ASCII
#   or IP type of value.
# Valid DHCP entry:
DHCP_CLIENT_MAC:  00:60:1D:02:0B:FE
CLIENT_IP:        135.3.94.136
SERVER_IP:        135.3.94.76
RLYAGNT_IP:       135.3.94.3
NETMASK:          255.255.255.0
GATEWAY:          135.3.94.1
SERVER_NAME:      server_name_here
BOOTFILE:         some_filename_here
STR_OPTNO_131:    some_ascii-string_here
HEX_OPTNO_132:    AABBCCDDEEFF
IPA_OPTNO_133:    4.8.12.16
STR_VSOPTNO_11:   ascii_string
HEX_VSOPTNO_132:  112233
IPA_VSOPTNO_13:   1.2.3.4
#
# Note that in the above example, NETMASK is the same as IPA_OPTNO_1
# and GATEWAY is the same as IPA_OPTNO_3
#
# Valid BOOTP entry:
BOOTP_CLIENT_MAC: 00:60:1D:02:0B:FC
```

```
CLIENT_IP:        135.3.94.131
SERVER_IP:        135.3.94.76
RLYAGNT_IP:       135.3.94.1
NETMASK:          255.255.255.0
GATEWAY:          135.3.94.1
SERVER_NAME:      server_name_here_too
BOOTFILE:         some_other_filename_here
# Default..
# Uncomment this entire entry if a default is to be specified.
# It is shown here for example purposes only.
#DHCP_CLIENT_MAC: 00:00:00:00:00:00
#CLIENT_IP:       135.3.94.148
#SERVER_IP:       135.3.94.76
#NETMASK:         255.255.255.0
#GATEWAY:         135.3.94.1
#SERVER_NAME:     servername_again
#BOOTFILE:        yet_another_filename_here
```

17.9 ELF

Extract information from or compress sections of an ELF file.

USAGE:

elf [-a:B:cfMm:P:sS:vz] {elf_filename}

DESCRIPTION:

This tool is one of three tools (coff, elf & aout) that basically perform the same tasks, but on different file formats. It allows the user to dump different portions of the file header to stdout in a readable format. Also, future plans include support for using zlib to compress each of the .text and .data sections so that TFS can store a compressed file in flash and decompress directly from TFS to the absolute-linked location for the application to run in.

OPTIONS:

- -a {filename}
  append file to end of -S file;
- -B {bin_filename}
  convert file (elf_filename) to binary (bin_filename) for transfer to absolute memory space;
  GNU equivalent: `objcopy --output-target=binary --gap-fill 0xff  elf_filename bin_filename`
- -c
  BE-to-LE convert... By default, this tool assumes the elf file was built with big-endian control structures. This option will convert to little endian control structures.
- -f
  show ELF file header;
- -M
  show ELF map with file offsets;
- -m
  show ELF map without file offsets;
- -P {pad-byte}
  by default, 0xff is used, this provides an override;
- -s
  show ELF section headers;
- -S {filename}
  strip to specified file
- -v
  verbose (debug) mode
- -V
  print the time/date the tool was built
- -z {zip level}
  compress the COFF sections using zlib compression level "zip level". Refer to TFS decompression for more details on this.

NOTES:

- If, after running this tool, some type of file error (read, write, lseek, etc...) occurs, it is likely that the executable being operated on was built with control structures of a different endian-type. Try re-running with the -c option.

EXAMPLES:

- elf -z6 app
  Compress the elf file "app" using zlib compression level 6. The output is placed in the file app.ezip.

EXIT STATUS:

0 if successful, else 1

17.10 F2MEM

Convert a series of files into a memory image for transfer to a flash device.

USAGE:

f2mem [a:B:b:ef:m:O:o:p:s:S:T:t:Vv]

DESCRIPTION:

f2mem (files to memory translator) is a tool that allows the user to take a group of files and generate a file that is suitable for a flash programmer. The intent is to feed it a monitor binary followed by a list of files that are destined to exist in TFS (Tiny File System). The monitor binary is placed at the base of the memory, then, starting at the offset specified by -t, each of the remaining files are processed and made to look as if they were residing in flash memory under TFS.

OPTIONS:

- -a {tfsname}
  ASCII file for TFS
  For all files that are destined for TFS, and are considered ascii files, this option is used. The conversion made here is to replace "\r\n" with "\r" in the ascii data.
- -B {address}
  address of flash base in CPU memory (default=0)
  This address is the actual address that the base of FLASH will be residing at when the device is part of the system CPU's memory map.
- -b {tfsname}
  binary file for TFS
  Similar to the -a option, but there is no conversion of any kind made on the file. It is placed in TFS just as it exists on the host.
- -e
  The default mode of operation for f2mem is to swap bytes in the TFS header structure because generally, it is assumed that f2mem is running on a little-endian machine but the file it is creating is destined for a big-endian machine. The -e option shuts off that swap.
- -f {hexbyte}
  byte to be used as filler (default=0xff)
  The default of 0xff is used so that all of the unwritten flash can be written (1-to-0 transition in flash space). If this is not the desirable case, then use this option to override the default.
- -m {filename}
  monitor file
  This is a required option, since this tool is assumed to be building a bootable memory image and the monitor is what does the booting.
- -O 's|b'
  Output type: S3-record or binary (default=S3).
- -o {filename}
  ASCII file for TFS
  output file (default = mem.bin | mem.srec).
- -p {pad-to size}
  Pad the file out to the specified size using the fill byte specified by -f as the pad (default is no padding).
- -s {m|a}
  Swap bytes in monitor binary (m) or all bytes (a).
- -S [address}
  s-record base address (default=0)
  In some cases, the S-record file will be used to program a device using a flash programmer that is external to the actual system that the flash will reside in. In this case, it is likely that the S-record base will be 0 because the addresses will be relative to the base of the flash device. In other cases, the S-record is used to program memory while it is part of the CPU's memory space, so the base address used in the S-record file will be relative to the CPU space and will typically be the same value used for the -B option.

- -T {#}
  TFS header version (default is 1).
- -t {###}
  Offset relative to base of flash for TFS offset
  This is a required option that tells f2mem where the beginning of TFS space is relative to the base of the flash.
- -V
  Show version.
- -v
  Enable verbosity.

NOTES:
- The -m and -t options are required.
- Syntax for tfsname: name,flags[,info]
- This tool assumes the monitor is the first piece of binary destined for the flash part.

EXAMPLES:
- **f2mem -v -Ob -m monppc.bin -a monrc,e -t 0x40000 -B 0x80000000**
  Build a memory image with the monitor binary being monppc.bin, and a monrc file as the only file in TFS. TFS starts at offset 0x40000 relative to the base of the flash and the base of the flash is at address 0x80000000 relative to the CPU's address bus. The output is binary, and stored in the file mem.bin. Note that the file specification of monrc contains a ",e" after it. This is the notation used by TFS to indicate that it is to be stored in TFS with the "e" flag set (executable). Note also that since monrc is an ascii file, it is an argument to the '-a' option.
- **f2mem -v -Os -o ias.srec -m monppc.bin -a monrc,e -b ias,eEB -t -x40000 -b 0x80000000**
  Build a memory image with the monitor, monrc file and an ELF based executable called "ias". The output is in S3-record format and will be stored in the file ias.srec. Note that the ias file is a binary file, so it is an argument to the -b option.
  f2mem -v -Os -o ias.srec -m monppc.bin -a monrc,e -t -x40000 -b 0x80000000 -S 0x80000000
  Build a memory image with monitor and monrc file, but set up the S-record so that the base address is relative to the CPU's memory space, not the FLASH device.

EXIT STATUS:
0 if successful, else 1

17.11 FCRC

Run the same CRC on a file that is used on the uMon based target.

USAGE:

fcrc {filename}

DESCRIPTION:

This tool simply allows the user to download a file to the target system and then verify that the memroy transfer was successful. In conjunction with the "-C" option of the "mt" command, a block of memory can be verified for sanity at any time after it is transferred to the target. The newmon command also outputs the CRC value of the binary file it is transferring; thus, allowing the user to run the "mt –C" command to verify at a later time that the transfer was successful.

OPTIONS:

- -V
  display the version (build date) of the tool

EXAMPLES:

- fcrc build_CSB472/boot.bin
  Print out the CRC32 of the file build_CSB472/boot.bin.

EXIT STATUS:

0 if successful, else 1

17.12 MACCRYPT

Convert a MAC address to an encrypted string.

USAGE:

maccrypt [-f:uV] "MAC Address"

DESCRIPTION:

This tool provides the "backdoor" access to a password protected MicroMonitor environment. Typically, this would be used if a MicroMonitor based system was configured with password protection but the users forgot the passwords.

OPTIONS:

- -f {filename}
  This option overrides the default data table used by the proprietary scrambler.
- -u
  Use UNIX crypt() instead of proprietary. Use of this option depends on whether the monitor was built to use the proprietary version or standard Unix version of crypt().
- -V
  display the version (build date) of the tool

EXAMPLES:

- maccrypt 00:60:1D:02:0B:FD
  Print out an encrypted string that will be accepted as a valid password by a target running MicroMonitor with that MAC address.

EXIT STATUS:

0 if successful, else 1

17.13 MAKE2FLIST

Build a file list based on the output of 'make'.

USAGE:

make2flist [options] {compile_cmd} {filename}

DESCRIPTION:

This tool is useful if you use cscope, ctags and/or any other tool that needs a list of the source files that are in your project. The idea is to simply parse the output of GNU make and from that, create a list of files that are part of the build. When building a large program (linux kernel in this case), redirect all output to stdout and also a file (using 'tee') as follows…

```
make ARCH=ppc CROSS_COMPILE=powerpc-405-linux-gnu- zImage.initrd 2>&1 | tee make.out
```

That creates a file that contains this kind of output, copied to the file 'make.out' (obviously only a portion of the output is shown)...

```
make[2]: Entering directory `/home/els/kernel/linux-2.4.22/kernel'
powerpc-405-linux-gnu-gcc -D__KERNEL__ -I/home/els/kernel/linux-2.4.22/include -
Wall -Wstrict-prototypes -Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -fo
mit-frame-pointer -I/home/els/kernel/linux-2.4.22/arch/ppc -fsigned-char -msoft-
float -pipe -ffixed-r2 -Wno-uninitialized -mmultiple -mstring -Wa,-m405   -nostd
inc -iwithprefix include -DKBUILD_BASENAME=sched  -fno-omit-frame-pointer -c -o
sched.o sched.c
powerpc-405-linux-gnu-gcc -D__KERNEL__ -I/home/els/kernel/linux-2.4.22/include -
Wall -Wstrict-prototypes -Wno-trigraphs -O2 -fno-strict-aliasing -fno-common -fo
mit-frame-pointer -I/home/els/kernel/linux-2.4.22/arch/ppc -fsigned-char -msoft-
float -pipe -ffixed-r2 -Wno-uninitialized -mmultiple -mstring -Wa,-m405   -nostd
inc -iwithprefix include -DKBUILD_BASENAME=dma  -c -o dma.o dma.c
```

Now run 'make2flist' on that output file...

```
make2flist -h powerpc-405-linux-gnu-gcc make.out >cscope.files
```

And it generates a file that looks like this…

```
/home/els/kernel/linux-2.4.22/kernel/sched.c
/home/els/kernel/linux-2.4.22/kernel/dma.c
```

The tool looks for two main items in the file: the "Entering directory" string and the specified compile command. Each time the "Entering directory" string is found, make2flist changes it's file prefix to the following text, then when a compile command string is found, make2flist outputs the prefix followed by the filename. It certainly makes some assumptions about the format of the make.out; however, generally speaking for GNU make, it works well.

OPTIONS:

- -h
  Include all header files in directories entered.
- -v
  Verbose mode.
- -V
  Display the version (build date) of the tool

EXIT STATUS:

0 if successful, else 1

17.14 MKUPDATE

Build a script that will query the target for updates.

USAGE:

mkupdate [options] {file1} [file2] [file3] … [fileN]

DESCRIPTION:

Based on a set of host-resident files, build a script that can be downloaded to a target and used to compare the files on the target to those on the host (using CRC). This allows a system to startup, run DHCP to retrieve this file, then run this file (quick script) to see if any target-resident files need to be updated. Then the bootfile can always be the same, but the content of the script can be used to update the files on the target.

In the nutshell, this tool creates a script that computes the crc of the file on the host and uses that computed value to compare with the file that is on the target. On the host, the 'fcrc' command can be used at the command line to compute a file's CRC, and on the target, the "mt –C" command can be used to compute (using the same CRC polynomial) the CRC. This script simply compares the crc that was computed on the host with the crc that is computed on the target to determine if the file needs to be downloaded. Refer to the example script (as it is generated by mkupdate) below…

OPTIONS:

- -d
  debug mode (for development only)
- -D {prefix}
  specify a first-level directory for tftp get
- -i
  use target's IP as first-level directory for tftp get
- -I {prefix}
  similar to -i, but include a second prefix
- -v
  Verbose mode.
- -V
  Display the version (build date) of the tool

EXAMPLE:

Use the mkupdate tool to create a script for updating the file called README…

```
mkupdate d2u.exe u2d.exe

#
#
if $TFTPSRVR seq \$TFTPSRVR goto NOSRVR_ERROR
set FTOT 0

#
##############################################
# File tests:
##############################################
#

# Test d2u.exe:
set FILE d2u.exe
tfs base d2u.exe BASE
if $BASE seq \$BASE gosub LOAD_d2u.exe
tfs size d2u.exe SIZE
if $SIZE seq \$SIZE goto DLDFAIL_ERROR
mt -C $BASE $SIZE
if $MTCRC ne 0x4d29a548 gosub LOAD_d2u.exe

# Test u2d.exe:
```

```
set FILE u2d.exe
tfs base u2d.exe BASE
if $BASE seq \$BASE gosub LOAD_u2d.exe
tfs size u2d.exe SIZE
if $SIZE seq \$SIZE goto DLDFAIL_ERROR
mt -C $BASE $SIZE
if $MTCRC ne 0x659289db gosub LOAD_u2d.exe

echo $FTOT files updated.
exit

#
##############################################
# Subroutines:
##############################################
#

#
# LOAD_d2u.exe:
tftp -F d2u.exe $TFTPSRVR get d2u.exe
set -i FTOT
return

#
# LOAD_u2d.exe:
tftp -F u2d.exe $TFTPSRVR get u2d.exe
set -i FTOT
return

#
##############################################
# Error handlers:
##############################################
#

# NOSRVR_ERROR:
echo This script requires the TFTPSRVR shell var be set.
exit

# DLDFAIL_ERROR:
echo Failed to download ${FILE}, script aborted.
exit
```

Notice in the script above that for each file (in this case d2u.exe and u2d.exe) there is a comparison section (see below) that compares the pre-computed CRC (as was done by mkupdate) with the actual file's CRC as is done with the "mt –C" command. If the CRCs don't match, then the appropriate routine is branched to and the file is updated using TFTP…

```
# Test d2u.exe:
set FILE d2u.exe
tfs base d2u.exe BASE
if $BASE seq \$BASE gosub LOAD_d2u.exe
tfs size d2u.exe SIZE
if $SIZE seq \$SIZE goto DLDFAIL_ERROR
mt -C $BASE $SIZE
if $MTCRC ne 0x4d29a548 gosub LOAD_d2u.exe
```

EXIT STATUS:

0 if successful, else 1

17.15 MONCMD

Interface to the monitor's UDP command interface port.

USAGE:

moncmd [-bil:mp:qrvVw:] {target ip-address} "target command string"

DESCRIPTION:

This tool allows the user to communicate with a target running MicroMonitor through Ethernet/UDP. The monitor has a server waiting for incoming command strings on port 777 (or whatever port is assigned in the MCMDPORT shell variable). It will process the command string sent via moncmd just as it would process a command from the console interface.

OPTIONS:

- -b
  Binary mode
- -i
  Interactive mode, for entering mutliple commands to one target.
- -l {##}
  Loop until interrupted (use ## millisecond delay between repetitions).
- -m
  Receive multiple responses.
- -p {##}
  Override default port number of 777;
- -q
  Quiet mode (don't print the timeout message);
- -r
  Retry if command-not-received. Note that if the command is received, but not completed the retry does not apply.
- -v
  Verbose mode (print the 'Sending...' string);
- -V
  Display the version (build date) of the tool
- -w {##}
  Number of seconds to wait for response, after which a timeout will occur (default is 10 seconds);

EXAMPLES:

- moncmd 135.3.94.136 "tfs ls"
  Send the command string "tfs ls" to a target at IP address 135.3.94.136
- moncmd -w0 135.3.94.136 "reset"
  Tell the target at IP address 135.3.94.136 to reset. Note that waittime is set to zero because there will be no response, since we are resetting the target firmware.

EXIT STATUS:

- 0 if successful
  Command was received, executed and completed.
- 1 if timeout waiting for command completion
  In this case, moncmd verified that the command was received by the target, but there was no command completion handshake. This would be the type of timeout that would occur if moncmd was used to issue a target reset.
- 2 if error
  Some kind of usage error typically.
- 3 if timeout waiting for command reception
  In this case, moncmd could not even verify that the command was received by the target.

17.16 MONSYM

Convert a file of tabulated data into the format used by MicroMonitor's command-line symbol retrieval.

USAGE:

monsym [-d:l:p:P:s:S:vV] {filename}

DESCRIPTION:

This tool is used on a file that was created by "nm" (or some equivalent). Most of what this is typically used for could also be done with a combination of awk & sort. It does some very simple column re-arranging of tabulated data in files. The monitor has the ability to process "symbols" (see discussion on shell variables and symbols). This is done through a file in TFS that is assumed to contain lines of data where each line is of the format:

```
{symbol} whitespace {data}
```

Typically, the output of nm or some other tool that generates a readable listing of symbols contains a lot of information that is of no value for basic symbol lookup. This tool simply extracts the data field and symbol-name field from each line of the input file and prints to stdout in the format above. This file can then be transferred to a target and used by the monitor for symbol lookup.

OPTIONS:

- -d {col #}
  Column number from which the data is to be extracted from the incoming file (default = 1);
- -l {size}
  Maximum size of an incoming line (default = 64);
- -p {data prefix string}
  A prefix attached to the data prior to output;
- -P {symbol prefix string
  A prefix attached to the symbol prior to output;
- -s {col #}
  Column number from which the data is to be extracted from the incoming file (default = 3);
- -S {x|d}
  Sort the output based on the content of the data column being hex (x) or decimal (d).
- -v
  Verbose (debug) mode
- -V
  Print the time/date the tool was built

EXAMPLES:

- monsym -p0x ias.sym
  From the incoming file ias.sym, output column 3 followed by whitespace and column 1. The data from column 3 is prefixed by 0x. For example if the ias.sym file contained...

```
00018000 T _sysInit
00018000 t gcc2_compiled.
0001805c t vxpbi1
0001809c t gcc2_compiled.
0001809c T sysSerialIntEnable
0001811c T sysSerialIntDisable
0001819c T sysSerialHwInit
00018308 T sysSerialHwInit2
00018364 T sysSerialChanGet
000183bc T sysSerialReset
```

then the result would be...

```
_sysInit 0x00018000
gcc2_compiled. 0x00018000
vxpbi1 0x0001805c
gcc2_compiled. 0x0001809c
sysSerialIntEnable 0x0001809c
sysSerialIntDisable 0x0001811c
sysSerialHwInit 0x0001819c
sysSerialHwInit2 0x00018308
sysSerialChanGet 0x00018364
sysSerialReset 0x000183bc
```

EXIT STATUS:
0 if successful, else 1

17.17 NEWMON

Automatically update a monitor's boot flash to a new version.

USAGE:

newmon -[A:B:b:p:uw:vV] {target ip-address} {binary filename}

DESCRIPTION:

This tool allows the user to upgrade a target's boot monitor. **WARNING!!!** This command will take the requested file and assume it is a binary file destined for the boot sector of the target device. If you give it the wrong file, you will destroy your boot sector!!! Refer to section 12.3 above for an example use of this tool.

OPTIONS:

Note that these options only apply to the .exe version. The script does not have any options.

- -A {###}
  override default use of $APPRAMBASE shell variable as location for tftp transfer and source for flash ewrite.
- -B {###}
  override default use of $BOOTROMBASE shell variable as target for flash ewrite
- b {#}
  override default flash bank of 0
- -p {##}
  override default port number of 777
- -u
  issue the 'flash unlock' command prior to 'ewrite'
- -w {##}
  number of seconds to wait for a target response, after which a timeout will occur (default is 10 seconds)
- -v
  verbose mode (print the 'Sending...' string)
- -V
  display the version (build date) of the tool

EXAMPLE:

- newmon 135.3.94.136 mon403.bin
  Place the content of mon403.bin into the boot flash of the target at address 135.3.94.136

EXIT STATUS:

Note that these exit status apply to the.exe version only.

- 0 if successful
  Command was received, executed and completed.
- 1 if timeout waiting for command completion
  In this case, moncmd verified that the command was received by the target, but there was no command completion handshake. This would be the type of timeout that would occur if moncmd was used to issue a target reset.
- 2 if error
  Some kind of usage error typically.
- 3 if timeout waiting for command reception
  In this case, moncmd could not even verify that the command was received by the target.

17.18 TNT

Provide a simple terminal connection (i.e. tty) with a telnet server back end.

USAGE:

tnt [options]

DESCRIPTION:

This tool is useful when you want to share interaction on a UNIX based serial port with multiple users that are not necessarily local. The idea is simple… provide a simple terminal connection (like minicom, but much less flexible at the moment), thus allowing the user of tnt to communicate with a device on a UNIX tty. The added feature is that tnt also runs a telnet server that allows other clients to connect and interact with that same tty. Note, if you're running with a Windows host, then you can use uCon[91] instead of this.

OPTIONS:

- -b ###
  baud rate (default is 9600)
- -n
  disable the telnet daemon.
- -h
  dump the help text.
- -p ####
  port number used by tenlet server (default = 8000).
- -P ******
  specify a password that would need to be entered by the client (default = none)
- -r
  telnet clients are read only (default = read/write)
- -t ****
  tty device name (default = /dev/ttyS0)
- -v
  show some verbosity
- -V
  show version (i.e. build date) of tool
- -w ****
  specify telnet client welcome message (default = none)

[91] uCon is a Windows-only program that is also part of the MicroMonitor package; however, it isn't distributed with the uMon tarball simply because it is a VCC-based program. It can be downloaded from http://www.microcross.com/html/micromonitor.html.

17.19 TTFTP

Run a tftp client or server.

USAGE:

ttftp [-ad:qr:t:T:vVx] {target ip_address} {get | put | srvr} [source] [destination]

DESCRIPTION:

This tool is similar to many other tftp clients,. It runs standard TFTP, but with a few added features that make it more useful. For example, the status of the download is available so that the user can determine if it is in progress, or hung. Also, a single-user server is built into the same executable. Note that the Solaris version of this does not currently support the 'srvr' capability.

OPTIONS:

- -a
  use netascii mode for 'put'; default is octet (binary);
- -d {per-packet-delay}
  per-packet-delay (milliseconds) inserted between each client-based outgoing packet .
- -q
  quiet mode (without this, ttftp will print one dot per block transferred);
- -r {rfc2349_tsize_off}
  use this to disable the client's use of the TSIZE extension as is specified in RFC 2349. This is applicable to "put" only.
- -t {timeout}
  inactivity timeout used by server; default is 60 seconds after which any transaction in progress is aborted.
- -T {test_setup_string}
  used for debugging tftp clients/servers.
- -v
  verbose mode (display details of each TFTP packet)
- -V
  display the version (build date) of the tool
- -x
  when ‘srvr’ is the argument, this option tells the server to terminate (exit) automatically after receiving one file.

EXAMPLES:

- ttftp srvr
  Run as a single-user server with the current working directory being relative to the directory in which the server was started. Note that this server will definitely get confused if multiple clients are thrown at it. Its purpose is of single-system lab testing only. Also note that this server option is not available on the SUN version of ttftp.
- ttftp -t 120 srvr
  Run as a single-user server with the current working directory being relative to the directory in which the server was started. Allow for up to 120 seconds of inactivity before timing out. Inactivity refers to an active transaction (WRQ or RRQ) but no data transfer.
- ttftp -a 135.3.94.136 put monrc monrc,e
  Copy the file named monrc (currently on the host) to TFS on the target at IP address 135.3.94.136. The file flags on the destination will be 'e' (executable), meaning that it is an executable script.
- ttftp -r rfc2349_tsize_off -a 135.3.94.136 put monrc monrc,e
  Same as above, but without the "tsize" extension.
- ttftp 135.3.94.136 get afile
  Retrieve the file 'afile' from the target and place it on the host with the same name.

EXIT STATUS:

0 if successful, else 1

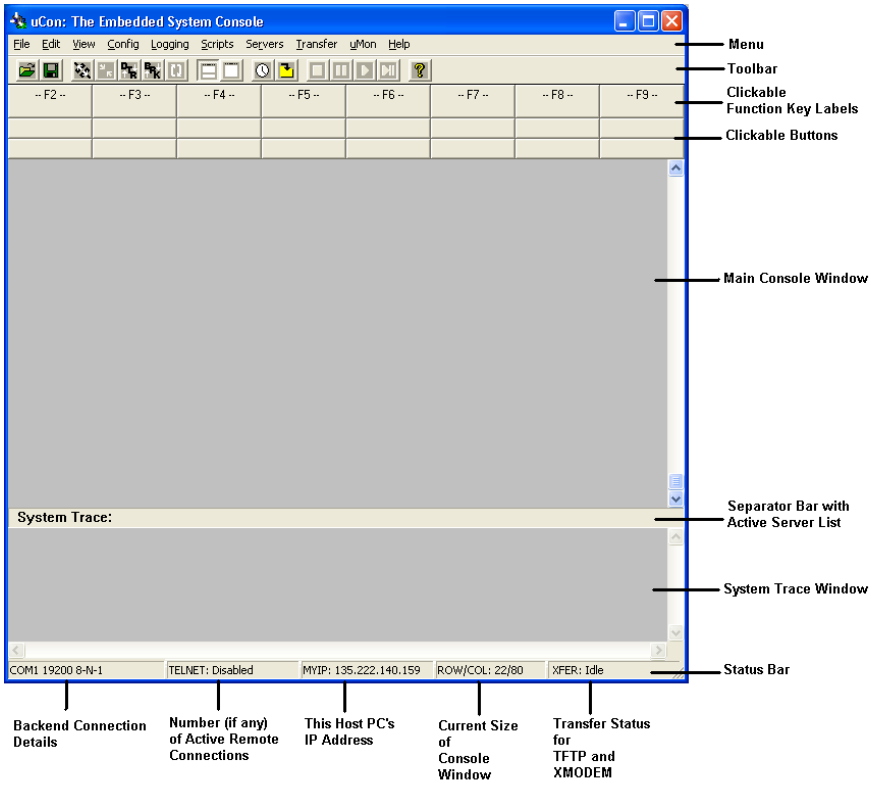
17.20 uCon

uCon, is a windows-only (for now) terminal emulator designed and written by an embedded system's devloper (me!) for use during embedded systems development. The tool provides terminal emulation capabilities for serial port, telnet, ssh and rlogin connections, plus a suite of capabilities that are very handy for the embedded systems development environment in general. The term "uCon" is *supposed* to imply "embedded systems console"; however, it is by no means limited to just embedded systems development.

Features include:

- Terminal emulation (VT100) for serial port or telnet-based devices.
- A terminal server mode allowing others (via telnet client) to simultaneously attach to the same device.
- Programmable function keys to invoke commands and/or scripts.
- Complete built-in help covering all topics.
- TFTP, FTP, SYSLOG, & DHCP/BOOTP servers.
- TFTP and XMODEM file transfer
- Per line timestamping
- Logging (standard and longterm)
- Versatile scripting language for interaction with the connected device and the user. The language includes conditional branching, interaction with the target as well as interaction with the user through configurable dialog boxes.
- Some uMon specific capabilities like moncmd, newmon and a convenient TFS file backup & restore.
- The ability to turn on trace-mode for all/any of the internal facilities.

This tool is not part of the MicroMonitor tarball. It is a windows-only application; hence, is built with Microsoft VCC only. It can be downloaded from the same Microcross site as the uMon tarball; however, does not include source. Here's a screenshot…


uCon: The Embedded System Console
File Edit View Config Logging Scripts Servers Transfer uMon Help
-- F2 -- -- F3 -- -- F4 -- -- F5 -- -- F6 -- -- F7 -- -- F8 -- -- F9 --
System Trace:
COM1 19200 8-N-1
TELNET: Disabled
MYIP: 135.222.140.159
ROW/COL: 22/80
XFER: Idle
Menu
Toolbar
Clickable Function Key Labels
Clickable Buttons
Main Console Window
Separator Bar with Active Server List
System Trace Window
Status Bar
Backend Connection Details
Number (if any) of Active Remote Connections
This Host PC's IP Address
Current Size of Console Window
Transfer Status for TFTP and XMODEM

17.21 VSUB

Variable substitution tool used with the monitor build process.

USAGE:

vsub {infile} {outfile} [TOKEN1=VALUE1] [TOKEN2=VALUE2] …

DESCRIPTION:

This tool provides a simple way to convert whitespace delimited tokens in one file into values in a second file. This could easily be done with sed as well; however under the context in which it is used in the monitor's makefile, this tool makes it much easier.

OPTIONS:

- -v
  Verbose mode
- -V
  display the version (build date) of the tool

EXAMPLES:

- vsub map.ldt map.ld BOOTROMBASE=0x100000 BOOTRAMBASE=0x30000
  Given the following map.ldt file…

```
MEMORY
{
      rom :        org = BOOTROMBASE, len = BOOTROMLEN
}
```

the command: `vsub map.ldt map.ld BOOTROMBASE=0x100000 BOOTROMLEN=0x7fff`
will produce the following map.ld file…

```
MEMORY
{
      rom :        org = 0x100000,    len = 0x7fff
}
```

This is used by the makefile to create different memory maps without the need to touch the linker file.

EXIT STATUS:
0 if successful, else 1

17.22 Some Handy Host-Based Scripts…

The host based tools discussed above allow you to create some handy scripts that automate some of the interaction that may otherwise be necessary when working with the target. These scripts are part of the umon source tree and can be found under umon_main/host/src/scripts.

17.22.1 TVI: target vi

This script automates the process of pulling up a file from the target, modifying it, and pushing it back down to the target. If the environment variable TARGET_IP is set, then it uses that as the IP address of the target; otherwise the user must specify the IP address of the target as the second argument. One can immediately see that 'vi' can be replaced by any editor (temacs, tnotepad, etc..)…

```
#!/bin/bash
# tvi: wrap editor (in this case vi) with tftp pull and push so that ASCII
# files on target can be edited on host…
usage()
{
    echo "Usage: tvi {filename_on_target}  [target ip address]"
    echo "(uses \$TARGET_IP if set)"
}

if [ $# = 1 ]
then
    if [ "$TARGET_IP" = "" ]
    then
        usage
        exit 1
    fi
elif [ $# = 2 ]
then
    export TARGET_IP=$2
else
    usage
    exit 1
fi

tmpfile=`mktemp`         # If mktemp isn't available just use 'xxx'.

ttftp $TARGET_IP get '.' $tmpfile
export fullname=`grep ^${1}, $tmpfile`

ttftp $TARGET_IP get $1 $tmpfile
if [ $? != 0 ]
then
    exit
fi

vi $tmpfile

ttftp $TARGET_IP put $tmpfile $fullname
rm $tmpfile
```

17.22.2 TLS: target list

Same idea as "tvi" above, but this simply uses "moncmd" to remotely run "tfs ls *"…

```
#!/bin/bash
#
# tls:
# Target 'ls'...
#
usage()
{
       echo "Usage: tls [target ip address]"
       echo "(uses \$TARGET_IP if set)"
}

if [ $# == 0 ]
then
       if [ "$TARGET_IP" = "" ]
       then
              usage
              exit 1
       fi
elif [ $# != 1 ]
then
       usage
       exit 1
else
       export TARGET_IP=$1
fi

moncmd $TARGET_IP "tfs ls *"
```

17.22.3 TCAT: target cat

Same idea as “tvi” above, but this simply uses “moncmd” to remotely run “tfs cat <filename>”…

```
#!/bin/bash
#
# tcat:
# Target 'cat'...
#
usage()
{
       echo "Usage: tcat {filename} [target ip address]"
       echo "(uses \$TARGET_IP if set)"
}

if [ $# == 1 ]
then
       if [ "$TARGET_IP" = "" ]
       then
              usage
              exit 1
       fi
elif [ $# != 2 ]
then
       usage
       exit 1
else
       export TARGET_IP=$2
fi

moncmd $TARGET_IP "tfs cat $1"
```